/北方民族大学学术文库/

二维和高维空间的分形图形艺术

李水根　赵翔鹏　编著

随书附赠光盘

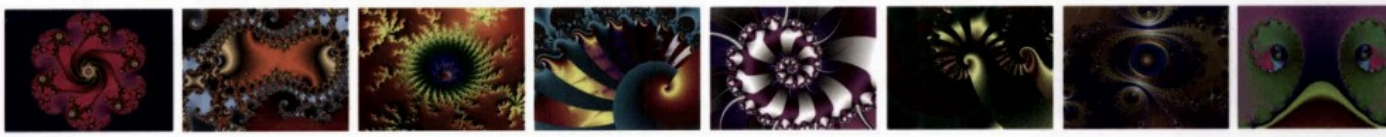


科学出版社
www.sciencep.com

(TP-4062.0101)

本书之特色

- **基础性** 分形之最基本知识
- **应用性** 用分形技术绘制大自然的山水花树及更多抽象图形
- **前沿性** 分型图形与混沌之关系
- **实用性** 大量绘画分形的程序及若干分形艺术创作软件介绍
- **完整性** 全部六种绘制分形图形之方法
- **欣赏性** 附带光盘有2000多张精美的彩色分形艺术图片供观赏
- **易用性** 附带光盘有几个可自由获取的优秀分形图形创作软件



ISBN 978-7-03-023256-4
9 787030 232564 >

科学出版社 技术分社
http://www.abook.cn

定价：45.00元(含光盘)

二维和高维空间的分形图形艺术

李水根　赵翔鹏　编著

科学出版社

北　京

内 容 简 介

本书系统介绍分形图形的艺术性和如何通过计算机对其进行创作。全书共 9 章，内容包含什么是分形、曼德勃罗（Mandelbrot）分形、茹利亚（Julia）分形、L-系统、迭代函数系统 IFS、四维空间的四元数分形、分形山地自然风景、各类奇异吸引子分形和著名分形创作软件的介绍和应用。

本书深入浅出，概念清晰，系统地阐述了分形图形的奇特性质和计算机在二维、三维和四维空间上的分形图形创作。

本书附有一张光盘，包括 7 个可自由获取的分形创作软件和程序、2000 多张包括全书插图的原彩色图像和其他彩色分形图片，供读者欣赏分形图形的神奇和美丽。

本书可供爱好分形学和计算机图形学的大学生和其他读者阅读和使用，亦可供美术设计爱好者阅读和使用。

图书在版编目（CIP）数据

二维和高维空间的分形图形艺术/李水根，赵翔鹏编著. —北京：科学出版社，2008

ISBN 978-7-03-023256-4

Ⅰ.二… Ⅱ.①李… ②赵… Ⅲ.分形理论-应用-计算机图形学 Ⅳ. TP391.41

中国版本图书馆 CIP 数据核字（2008）第 166237 号

责任编辑：鞠丽娜 / 责任校对：刘彦妮

责任印制：吕春珉 / 封面设计：三函设计

科学出版社 出版

北京东黄城根北街 16 号

邮政编码：100717

http://www.sciencep.com

新蕾印刷厂 印刷

科学出版社发行 各地新华书店经销

*

2009年1月第 一 版 开本：787×1092 1/16

2009年1月第一次印刷 印张：16 1/4 插页：1

印数：1–3 000 字数：382 300

定价：45.00 元（含光盘）

（如有印装质量问题，我社负责调换〈环伟〉）

销售部电话 010-62134988 编辑部电话 010-62135763-8002

版权所有，侵权必究

举报电话：010-64030229；010-64034315；13501151303

前　言

每个人都曾在少年时代上过美术课。但若干年过去了，喜欢各类美术作品的大有人在，但又有多少人能在宣纸上自己挥笔画出还算不错的作品来呢？更不用说“栩栩如生”、“形象逼真”之类的高品位画作了。笔者也是这样一个人，虽然喜欢看各类画展，但却也只能欣赏别人的画作而已，自己是无论如何也画不出一件稍微像样的美术作品。

20 世纪 90 年代，笔者在加拿大做访问学者的时候看到了一本名为“The Science of Fractal Images”的书。我一下子就被书里面的彩色插图迷住了。天哪，人居然能使用计算机画出如此美妙绝伦的人所不能想象的分形图像和栩栩如生、形象逼真的山水风景画来！有些自然风景画看起来简直就像摄影照片，完全可以乱真。从此我就有通过电脑来画画以至于弥补我在美术上先天不足的想法。在以后的日子里，笔者在西安交通大学长期为研究生和高年级本科生开设“分形”课程，并发现最能引起学生浓厚兴趣的是那些“计算机分形图形学”的内容。虽然在笔者的两本分形教材里已有一些这方面的内容，但那只能作为整个分形学体系的一部分，哪能把如此丰富多彩的分形图形艺术介绍无遗呢？况且该门学科还在快速发展之中，特别是三维艺术创作。经过较长时间的考虑，我和赵翔鹏博士（微软亚洲工程院）决定写一部内容比较详尽的分形图形艺术书籍。之所以加进“艺术”二字，实在是某些分形图像如曼德勃罗（Mandelbrot）集所含的艺术成分太丰富了，而且用分形理论去创作山水风景画本身就是一门艺术。这本书还有别于正规的“计算机分形图形学”，它不要求读者具有较高的数学知识，一般高中文化程度以上的计算机爱好者就能读懂这本书。当然在书的某些地方会出现一些涉及高等数学的内容，但读者完全可以跳过去，并不会影响对整体概念的理解。我们希望读者通过本书的阅读，能用计算机画出多姿多彩的分形图像，并能运用已有的分形绘图软件创作出高水平的艺术作品来。

分形艺术是分形学的一个组成部分，就当今而言，分形艺术创作者的兴趣主要趋向于下列几个方面：

1）复数多项式映射，这反映在本书的第 2 章和第 3 章；

2）四元数分形，本书第 6 章；

3）迭代函数系统，本书第 5 章；

4）分形山地自然风景，本书第 7 章；

5）L-系统，本书第 4 章；

6）奇异吸引子，本书第 8 章；

7）分形音乐，它是听觉艺术，不在本书视觉艺术写作之列。

本书共分 9 章。第 1 章是引论，主要是为不熟悉分形的读者写的，相信看过这一章的读者一定知道什么是分形了。

第 2 章介绍著名的曼德勃罗集。本章与其他章的内容描述有较大的区别，它不仅介绍曼德勃罗集的由来和算法，还花费很多的笔墨去分析神秘的曼德勃罗集的细微结构及

其丰富的艺术内涵。曼德勃罗集是分形世界中最著名的集合，是计算机视觉艺术中的极品。分形学家和动力系统专家对它的研究从未间断过。本章将把读者引入这神秘的世界，在惊叹它特殊魅力的时候，去欣赏它的无尚艳丽。在介绍完二维曼德勃罗集和二维广义曼德勃罗集之后，又叙述了几种伪三维曼德勃罗集，它们把本已很娇艳的平面图像演化到更令人惊叹的现实世界。说其是伪三维，是因为它并不是由平面数学公式在三维空间中的扩展去实现，而是由其他技术将其立体化而成。真正的三维曼德勃罗集将在第 6 章叙述。

第 3 章介绍与曼德勃罗集齐名的茹利亚集。如果说曼德勃罗集以其神秘莫测和幻想丛生而闻名的话，那么茹利亚集则以变化多端且无比妖娆而著称。说其变化多端，实乃它有无穷多的化身。除有大量的广义茹利亚集之外，还有像牛顿（Newton）分形、顾莫乌斯基-米拉（Gumowski-Mira）分形和 λ 分形等，这些都属于茹利亚分形范畴。有意思的是，无论是曼德勃罗集还是茹利亚集居然都源于同一个复数数学公式 $z = z^2 + c$ 的迭代，只是初值和常数的选择差异而已。本章在介绍上述缤纷现象后，也介绍了其伪三维茹利亚集，真正的三维茹利亚集也在第 6 章讲述。

第 4 章是 L-系统。二维 L-系统已有几本书介绍了，这里的内容更为丰富，也举了更好的例子。三维 L-系统在国内尚属少见，本书对其做了较多的叙述。当然除了在美术上的应用，L-系统在生物科学上应该有用武之地。

第 5 章讲的是迭代函数系统 IFS。IFS 在分形学中是很重要的一部分，它不仅可以绘画，还是分形图像压缩的理论根据。本书在叙述它的时候，尽量避免较深的数学背景，以便更多的读者能容易阅读。对 IFS 有更大兴趣的读者可参阅笔者在参考文献中列出的另两本书，那里对 IFS 有更多的数学描述，并揭示了 IFS 更本质的东西。虽然本书缺乏对数学的推理，但对 IFS 做图编码的叙述极为详尽，并总结出各种各样的编码方法，这是一般书籍所不及的。三维 IFS 是本章的重要亮点，它是 20 世纪 90 年代以后才发展起来的。不管在理论上还是实践方面都大有探索之处。从本质上来说，它应该是三维仿射变换，但直接从三维仿射变换编程并配之以三维成像系统则不是一件容易的事。本书介绍的编程语言特别简单，它完全建立在二维 IFS 码的基础上，再加立体空间的旋转和移动就可以画出三维的分形图像了。当然也有直接从三维仿射变换编码编程的，只是其 IFS 码更复杂，本书没有更多地涉猎这方面内容。

第 6 章是四元数（quaternion）分形。四元数是四维空间的超复数，由爱尔兰大数学家哈密尔顿于 1843 年发明。虽然它在其他学科中有所应用，但在分形学中画图却是 20 世纪 80 年代后期的事。正因为四元数在分形中的应用，才诞生了真正的三维曼德勃罗集和茹利亚集。当然也会有广义四元数曼德勃罗集和茹利亚集。由于四元数的运算我们目前（至少是从普遍意义上讲）还只限于加减乘除的四则运算，所以其广义集合这里只能介绍与之相关的部分，比如高次多项式分形、有理函数分形等，其中四元数牛顿分形集则是由我们自己独立创作出来的。

第 7 章是分形图形学中极为精彩的部分，即分形山地风景绘制。分形风景的创作理论除分形的基本理论外，主要依据是随机布朗运动及其演化出来的随机中点位移法。本章除了介绍基本的创作理论外，还特意解剖了一种叫随机“钻石-方块”的创作结构，

最后用 Java 语言做了绘画分形山地风景的程序分析。必须指出，山地风景画的创作是非常复杂的，仅用简单的理论方法是难以画出逼真的自然山地风光的。当今世界已有相当成熟的 Fractal landscape 绘图软件，用于编制程序时几乎考虑了自然界的方方面面，如山峦的起伏不平、草地的陪衬、海平面的高低、水的透明度、云彩的分布和阳光的照射等。这方面的内容在第 9 章将有更详细的分析。

第 8 章是奇异吸引子。奇异吸引子来源于对某些大自然现象模拟方程组的求解，这种解的集合实际上就是分形。虽然它不是前面所列的六大方面之一，但由于人们在解决科学与工程问题上经常遇到这种怪异而又绚丽多彩的图像，因此它实际上已成为分形图像艺术不可分割的一部分。对于奇异吸引子，国内有多本关于分形的书籍都有介绍，但介绍的种类很少。本章涉及的奇异吸引子数量超过七大类十多种，既有传统的洛伦兹（Lorenz）吸引子和亥农（Hénon）吸引子，还有形形色色的多元多项式吸引子和皮柯佛（Pickover）吸引子等。这些吸引子的图案极其漂亮，给人以赏心悦目的感受。

第 9 章介绍若干分形制图软件，既有很好用的 IFS 软件，又有综合性软件 Chaopro 和 fe（fractal explorer）。特别推荐的是分形山水风景发生器 Terragen（Terrain Generator）。本章用很大篇幅予以对软件进行详细说明。读者将看到，用该软件制作的山水风景画十分逼真，与摄影作品相比几乎可以乱真。

本书附带了一张光盘，里面除了全部插图 338 张（82%为彩色图片）以外，还有 1700 张左右彩色分形图片供读者欣赏，同时还给出 4 个分形图片制作软件。

本书的出版得到北方民族大学出版基金和信息与计算科学学院的资助，特在此表示衷心的感谢。

笔者虽然有长时期的“分形”课教学经历，但毕竟学识有限，不足与错误之处难免存在，恳请读者批评指正。

本书的一些插图与程序来自曹斌同学（香港科大博士生）和北方民族大学的杨晓通同学（已毕业），笔者在此谨致谢意。

李水根
北方民族大学信息与计算科学学院
2008 年 6 月

目　录

第 1 章　什么是分形 …… 1

第 2 章　曼德勃罗（Mandelbrot）分形 …… 7

2.1　逃逸时间法 …… 7

2.2　曼德勃罗分形的算法及其维数 …… 8

2.3　曼德勃罗分形结构的复杂性和艺术性 …… 15

2.3.1　顶部 …… 15

2.3.2　侧边点邻域 …… 16

2.3.3　曼德勃罗半岛 …… 16

2.3.4　边境点邻域 …… 18

2.3.5　半岛顶点邻域 …… 19

2.3.6　折叠点邻域 …… 21

2.4　广义曼德勃罗分形 …… 25

2.4.1　从变化映射 z^2+c 的幂次而得到的广义曼德勃罗集 …… 25

2.4.2　其他函数代替 z^2 的广义曼德勃罗集 …… 26

2.5　伪三维曼德勃罗分形 …… 28

第 3 章　茹利亚(Julia)分形 …… 34

3.1　茹利亚集的算法及其维数和性质 …… 34

3.2　广义茹利亚集 …… 44

3.2.1　整数次幂的广义茹利亚集 …… 44

3.2.2　其他函数代替 z^2 的广义茹利亚集 …… 46

3.2.3　牛顿分形 …… 48

3.2.4　米拉分形 …… 52

3.2.5　λ分形 …… 54

3.3　伪三维茹利亚分形 …… 55

第 4 章　L-系统 …… 58

4.1　二维 L-系统 …… 58

4.1.1　简单的向前生成元格式 …… 58

4.1.2　左右生成元的混合格式 …… 62

4.1.3　分枝结构的简单进退格式 …… 65

4.1.4　分枝结构带空指令的进退格式 …… 67

4.1.5　随机 L-系统 …… 68

4.2　三维 L-系统 …… 69

第 5 章　迭代函数系统 IFS …… 80

5.1　IFS 的算法 …… 80

5.1.1　确定性算法 …… 80

5.1.2 非确定性算法 ····· 84
5.2 IFS 的编码 ····· 89
5.2.1 解线性方程组的编码法 ····· 90
5.2.2 旋转和位移变换的编码法 ····· 90
5.2.3 利用拼贴方法的 IFS 编码 ····· 96
5.3 带参量的 IFS ····· 99
5.4 三维 IFS ····· 101
5.4.1 无厚度的三维图像生成 ····· 103
5.4.2 三维图像的 IFS 生成 ····· 104
第 6 章 四维空间的四元数分形 ····· 113
6.1 四元数基本理论 ····· 113
6.2 三维立体绘图的基本原理 ····· 116
6.3 四元数分形 ····· 118
6.3.1 四元数曼德勃罗集 ····· 118
6.3.2 四元数茹利亚集 ····· 120
6.3.3 四元数多项式和有理式的广义茹利亚分形 ····· 129
6.3.4 四元数牛顿分形 ····· 132
6.3.5 四元数λ分形 ····· 135
第 7 章 分形山地自然风景 ····· 137
7.1 布朗运动与随机中点位移法 ····· 137
7.2 分形山地风景的基本算法 ····· 142
7.2.1 矩形域 RMD 剖分法 ····· 142
7.2.2 影响作图的若干因素 ····· 143
7.3 随机钻石-方块算法 ····· 149
7.3.1 一维中点变换 ····· 150
7.3.2 钻石-方块算法 ····· 152
7.4 分形山地的 Java 程序设计 ····· 156
7.4.1 地形地图与分形地形 ····· 156
7.4.2 用马赛克技术处理地形 ····· 162
7.4.3 光线和阴影 ····· 163
7.4.4 观测点计算和变换世界 ····· 169
7.4.5 空间旋转与四元数 ····· 171
7.4.6 空间的转化 ····· 173
7.4.7 屏幕上的绘制与收尾工作 ····· 174
第 8 章 奇异吸引子 ····· 187
8.1 洛伦兹吸引子 ····· 187
8.1.1 洛伦兹 I 型 ····· 187
8.1.2 洛伦兹 II 型 ····· 191

8.1.3 洛伦兹 84 ····· 192
8.2 罗斯洛吸引子 ····· 192
8.3 杜玢吸引子 ····· 196
8.4 二元多项式吸引子 ····· 197
8.4.1 亥农吸引子 ····· 198
8.4.2 二元高次多项式型 ····· 205
8.5 三元多项式吸引子 ····· 208
8.5.1 二次多项式映射 ····· 208
8.5.2 三类参数量相异的二次型 ····· 210
8.6 皮柯佛奇异吸引子 ····· 214
8.7 其他类奇异吸引子 ····· 217
8.7.1 彼得·德·戎型 ····· 217
8.7.2 对称型 ····· 217
第 9 章 若干分形图形艺术软件介绍 ····· 220
9.1 IFS V1.41 ····· 220
9.2 ChaosPro3.2 介绍 ····· 222
9.2.1 简介 ····· 222
9.2.2 各种类型分形的参数设置 ····· 225
9.3 综合绘图软件 fe ····· 231
9.4 分形山地风景软件 Terragen ····· 234
9.4.1 主界面介绍 ····· 234
9.4.2 地表景色 ····· 236
9.4.3 水面、云彩和光照 ····· 241
附录 光盘内容 ····· 246
主要参考文献 ····· 247

第 1 章　什么是分形

在初等几何学里，一个几何体通常是由一个数学表达式来实现的，且这种表达式与该几何体的维数有关系。比如平面上一个点 M_0，通常它的数学表达式是：

$$M_0:\begin{cases}x=c_1\\y=c_2\end{cases} \tag{1.1}$$

式中 c_1 和 c_2 是常数。即一个点 M_0 无法用一个简单的数学方程式 $M_0=f(\cdot)$ 来表示，它的几何维数为零。对于平面上一条直线或光滑曲线 y，它可以由一个函数变量 x 来表达，即

$$y=f(x) \tag{1.2}$$

所以它是一维的几何体。对于立体空间的一张平面或光滑曲面 z，它能表达为

$$z=f(x,y) \tag{1.3}$$

即由两个变量表示，故其几何维数为 2。

这种几何维又称几何体的拓扑维。

上面说的维数是由几何体的函数变量来表示的，那么这个维数如何与几何体本身的拓扑结构发生关系呢？

一条长为 1 的直线段，如果将它 N 等分，每等分小线段长度为 r，于是有

$$N\cdot r=1 \tag{1.4}$$

或等价写为

$$N\cdot r^1=1 \tag{1.4'}$$

对面积为 1 的正方形做 N 等分，每个小正方形的边长为 r，则

$$N\cdot r^2=1 \tag{1.5}$$

对体积为 1 的正方体做 N 等分，每个小正方体的边长为 r，则

$$N\cdot r^3=1 \tag{1.6}$$

上面三个等式中，r 的幂次实际上就是该几何体的空间维数 D，于是我们有如下公式

$$N\cdot r^D=1$$

两边取对数，我们得到空间维数 D 的表达式：

$$D=-\frac{\ln N}{\ln r} \tag{1.7}$$

前面的几何空间维数与几何体本身的关系描述见图 1.1，但并不是所有的曲线或曲面都能由数学函数表达的。比如股票曲线难以由 $y=f(x)$ 来表示，地上的树木花草可以用 $z=f(x,y)$ 来描述吗？实在太难。我们不妨举两个例子来看看。

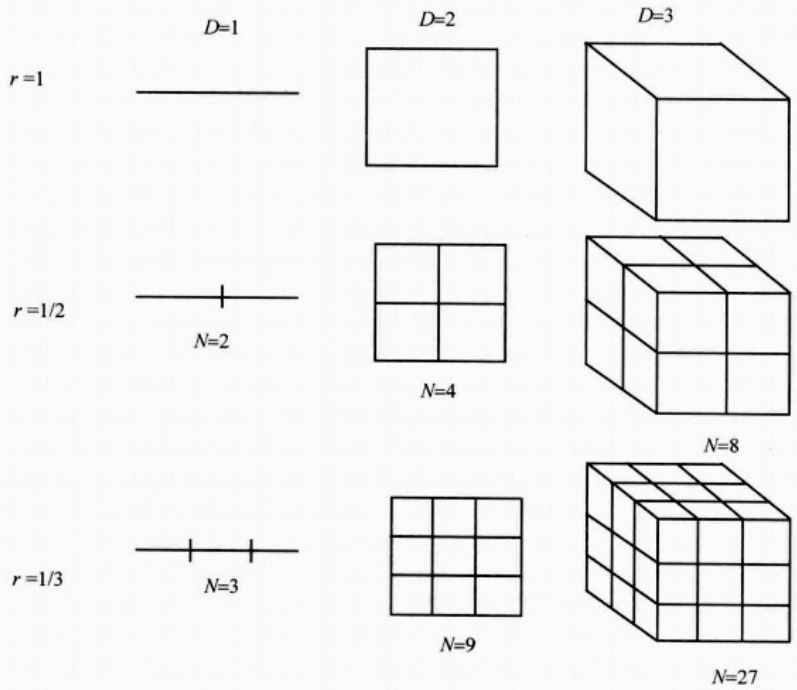


图 1.1　几何体与维数

例 1.1　冯·柯克（Von Koch）曲线。

1904 年，瑞典数学家冯·柯克设计了一条被称之为柯克曲线的图形，其设计步骤如下：设 E_0 为单位区间[0，1]，第一步，即 n=1，以 E_0 的中间三分之一线段为底，向上作一个等边三角形，然后去掉区间（$\frac{1}{3}$，$\frac{2}{3}$），得一条四折线段的多边形 E_1。第二步，即 n=2，对 E_1 的四条折线段重复上述过程，得一条十六折线段多边形 E_2。再重复上述过程，由 E_n 到 E_{n+1}，当 n 趋于无穷时，便得一条柯克曲线，它是一条处处连续但非常曲折的曲线。基本过程见图 1.2。

程序 1.1 给出一个用 C++递归法编的产生冯·柯克曲线的程序，供参考。

柯克曲线无法用简单的函数 $y=f(x)$ 来描述[我们强调是简单函数，是因为个别类似柯克曲线的图形，如维尔斯特拉斯（Weierstrass）曲线可以用含一个变量的无穷级数来表示]。它是一条处处连续的曲线，但极其粗糙，用一句高等数学的话来说就是点点不可微。虽然它是一条连续曲线，但其几何维数未必是 1。分析如下：

当 n=1 时，它有 4 条线段，每段长度为 1/3，n=2 时，有 16（$=4^2$）条线段，每段长度为 1/9($=1/3^2$)，显然到第 n 步时，总段数为 $N=4^n$，每段长 $r=1/3^n$。对照公式(1.7)，其几何维数为

$$D=\frac{\ln 4^n}{\ln 3^n}\approx 1.26186$$

于是一个令人震惊的事发生了：居然一条连续曲线的几何维数不是 1，而是大于 1！

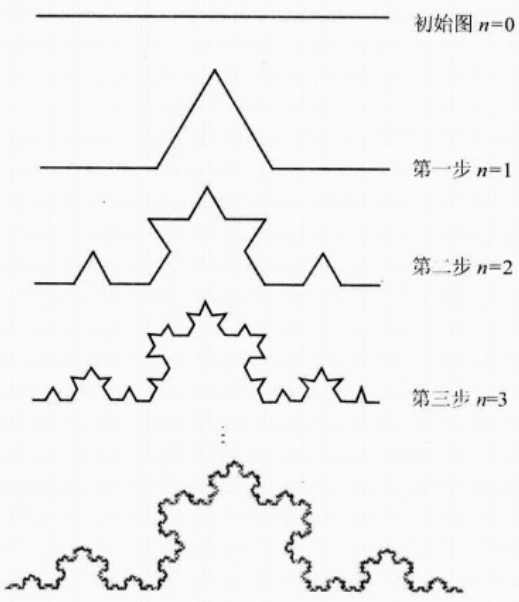


图 1.2　冯·柯克曲线的生成

例 1.2　希尔宾斯基（Sierpinski）三角形。

由波兰数学家希尔宾斯基命名的三角形，又称希尔宾斯基筛，希尔宾斯基垫片等。令 S_0 为边长为 1 的等边三角形，第一步（n=1），连接三条边的中点，得到四个全等三角形，去掉中间一个，保留其余三个得 S_1。第二步（n=2），对 S_1 的三个三角形重复刚才步骤得 S_2，它含有 9 个小三角形。如此重复上述步骤得 S_n，当 n 趋于无穷大时，便得希尔宾斯基三角形（见图 1.3 和图 1.4）。

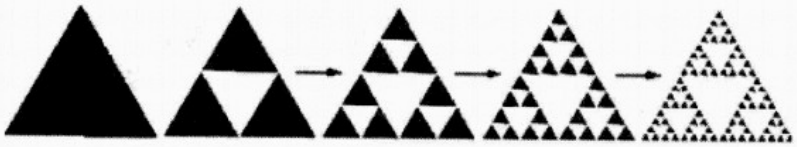


图 1.3　希尔宾斯基三角形的形成

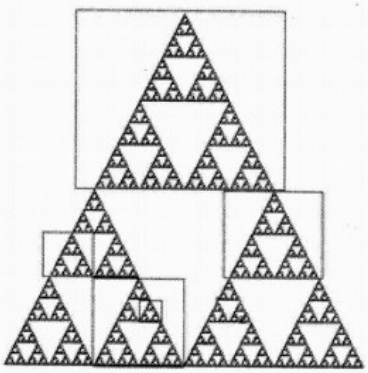


图 1.4　希尔宾斯基三角形

对于希尔宾斯基三角形，在第 n 步时，所有全等三角形的边数为 3^{n+1} 个，每边长度为 $\left(\frac{1}{2}\right)^n$，所以希尔宾斯基三角形的维数为

$$D=\frac{\ln 3}{\ln 2}\approx 1.58496$$

像柯克曲线和希尔宾斯基三角形这样的几何体，历史上曾被称作数学怪物（当然那时还未认识到其几何维数的非整数性）。现在知道它们实际上就是分形。

什么是分形？分形学科的创始人曼德勃罗（Mandelbrot）曾经用“几何维数大于其拓扑维数的集合”来定义，但终究不是很确切，那么究竟什么是分形呢？应该说，到目前还未有严格的定义。我们不妨引用法尔克内（K.Falconner）对分形集合 F 的描述：

1）F 具有精细的结构，即是说在任意小的尺度之下，它总有复杂的细节；

2）F 是如此地不规则，以至它的整体和局部都不能用传统的几何语言来描述；

3）F 通常具有某种自相似性，这种自相似性可以是近似的，也可能是统计意义上的（见图 1.4～图 1.7）；

4）F 在某种意义下的分形维数通常都大于它的拓扑维数；

5）在多数令人感兴趣的情形下，F 以非常简单的方法定义，或许以递归或迭代过程产生。

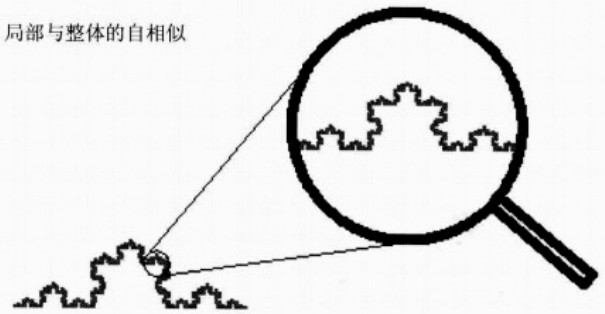


图 1.5　柯克曲线的自相似性

图 1.5 显示了柯克曲线局部与整体的相似性，这是一种严格的自相似。但现实中有大量并不具备严格自相似的几何体。比如人身体内血管，从大动脉到小动脉，从小动脉到静脉，从静脉到微血管，如此等等，其整个结构呈现分形特性。但微血管不可能无限分下去，这就是统计意义上的自相似性。再如现实中的树，从大树干到大枝叉，从大枝叉到小枝叉……虽然很相似，但毕竟是近似的。

大自然中的确有分形。图 1.6 是地球某一部分逐步放大后居然与原来图形极其相似。

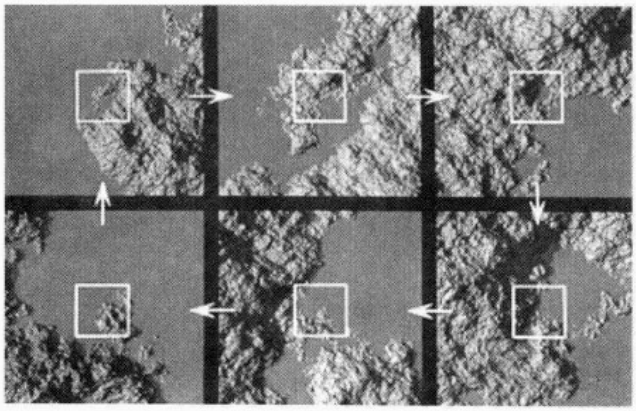


图 1.6　逐步放大后的相似性

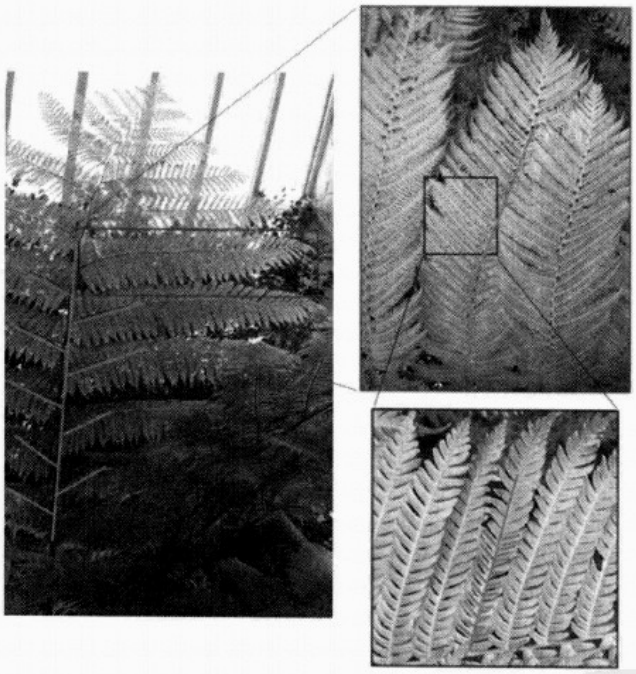


图 1.7　树叶的自相似性

图 1.7 中的植物叶子，也具备自相似性。

从几何体的维数来说，地球上的分形就更多了。海岸线的维数在 1.05～1.25 之间，山脉表面的维数大于 2，海浪起伏的海洋曲面维数也大于 2，等等。

分形几何学可以说是大自然几何学。大自然是何等地瑰丽！故而可以模拟大自然景色的分形艺术当然是璀璨无比的。

本书附带光盘中的第 1 章中，有 300 多张各种分形图片供欣赏。这些图片一般是早期制作的，其中多数由软件 FRACTINT 在 DOS 下产生。

程序 1.1　冯·柯克（Von Koch）曲线。

```
#include <stdio.h>
#include <graphics.h>
#include <complex.h>
#include <conio.h>
#include <math.h>
#define MAX 1024
complex a[MAX];
int num=5;
Koch (num)                    // calculate ptcoordinate
{
  int i;
  complex t[MAX];
  if (num==0) a[0] = complex (-300,0) ,a[1] = complex (300,0) ;
  else
  {
      Koch (num-1) ;
      for (i = 0;i <= pow (4,num-1) ;i++)
          t[i] = a[i];
      for (i = 0;i <pow (4,num-1) ;i++)
      {
          a[i*4] = t[i];
          a[i*4+1] = t[i]+ (t[i+1]-t[i]) * (1.0/3) ;
          a[i*4+3] = t[i]+ (t[i+1]-t[i]) * (2.0/3) ;
          a[i*4+2] = t[i]+ (t[i+1]-t[i]) *complex (0.5, sqrt (3) /6) ;
      }
      a[i*4] = t[i];
  }
}
complex CH (complex pt)     // changecoordinate
{return(complex( real(pt)+0.5*getmaxx(),-imag(pt)+0.5*getmaxy()));}
void draw (num)
{
  int i;
  for (i = 0;i<pow (4, num) ;i++)
      line (real (CH (a[i])) ,imag (CH (a[i])) ,
      real (CH (a[i+1])) ,imag (CH (a[i+1]))) ;
}

void main ()
{
  int i,gd, gm;
  gd=DETECT;
  setbkcolor (1) ;
  setcolor (14) ;
  initgraph (&gd, &gm, "") ;
  for (i = 0;i <= num;i++)
  {
      Koch (i) ;
      clearviewport () ;
      draw (i) ;
      getch () ;
  }
  closegraph () ;
}
```

第 2 章　曼德勃罗（Mandelbrot）分形

2.1　逃逸时间法

本章的后面几节作图法都与一种叫“逃逸时间法”的作图方法有关，而逃逸时间法是基于迭代法的一种画图法。“迭代”是一个古老的数学概念，这里我们要给出明确的定义。

定义 2.1　设 f 是一个变换，f 的向前迭代变换 f^n 定义为

$$f^0(x)=x,\quad f^1(x)=f(x),\quad f^{n+1}(x)=f(f^n(x)),\quad n=0,1,2,\cdots$$

例 2.1　令 $f(x)=\sin x$，若取 x=1.5，则

$$f(x)=0.9975,\quad f^2(x)=0.8401,\quad f^3(x)=0.7447,\quad \cdots,\quad \lim_{n\to\infty} f^n(x)=0$$

这是一个可预料的结果。

例 2.2　令 $f(x)=\cos x$，仍取 x=1.5，则

$$f(x)=0.0707,\quad f^2(x)=0.9975, f^3(x)=0.5424,\quad \cdots,\quad \lim_{n\to\infty} f^n(x)=0.739085,\quad \cdots$$

这是一个不可预测的结果。

设 $z\in C$，其中 C 是全体复数集合。令 (a,b) 和 (c,d) 分别表示坐标图上的左下方和右上方闭区域 W 的两个点（图 2.1），M 是正整数，定义 W 中的点

$$z_{pq}=(a+p\frac{c-a}{M},b+q\frac{d-b}{M}),\quad p,q=0,1,\cdots,M$$

实际操作时，用计算机屏幕上一个像素来表示这个点，观察轨道 $\{f^n(z_{pq})\}_{n=0}^{\infty}$ 上点的运行状况。

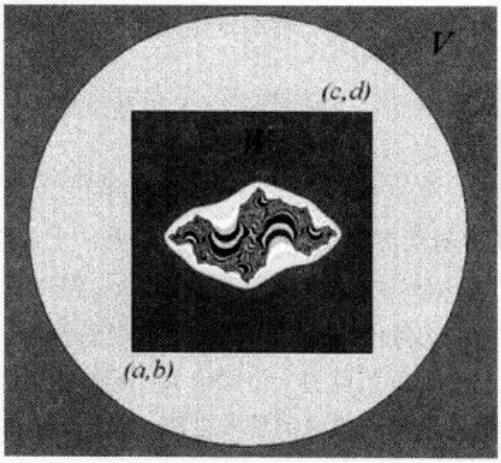


图 2.1　闭域

设 R 为正数，以原点为中心，R 为半径的圆包含了 W，定义

$$V=\{z\in C:|z|>R\}$$

令 N 为正整数，对每个 p,q=0，1，…，N 计算轨道 $\{f^n(z_{pq})\}$ 时，n 不能超过 N。如果当 n=N 时，$\{f^n(z_{pq})\}$ 的轨道点集没有落入 V，则转向下一个 (p,q) 值，否则当 $n\leqslant N$ 时，存在第一个整数 n 使 $f^n(z_{pq})\in V$，则相应于 z_{pq} 的像素点就要被点上一个颜色，然后的计算转向下一个 (p,q)。显然这种计算机画图方法提供了 W 中不同点轨道到达区域 V 的一个长度。

上面这种画图算法称为逃逸时间法。可理解为闭域 W 中的点轨道随时间变化是否逃出该区域。

下面给出一个用逃逸时间法画希尔宾斯基三角形的 Basic 程序。

程序 2.1　逃逸时间算法（注："'" 号后面是说明语句）。

```
NN=20: a=0:b=0:c=1:d=1:M=100'定义视窗 W 和 M
R=200 '定义 V 域
screen  9: cls
for p=1 to M: for q=1 to M
x=a+ (c-a) *p/M: y=b+ (d-b) *q/M '轨道集 z (p,q)
for n=1 to NN '在轨道 z (p,q) 上最多计算 NN 个点
if y>0.5 then x=2*x:y=2*y-1
else if x>0.5 then x=2*x-1:y=2*y
else x=2*x:y=2*y
end if
if x*x+y*y>R then '如果当前计算的点落在 V 内
  pset (p,q) ,n'在像素 z (p,q) 上着色 n,转向下一个 (p,q)
end if
if instat then end'按任意键停止计算
next n: next q: next p
end
```

2.2　曼德勃罗分形的算法及其维数

1980 年，曼德勃罗给世人提供了一幅无与伦比的杰作：曼德勃罗集。如图 2.2 所示，其创作过程如下：

令 $f(z)=z^2+c$，其中 $z,c\in C$，z 是复变量，c 是复常数。对变换 f 施行逃逸时间法，并得到如下迭代公式：

$$z_{n+1}=z_n^2+c_{pq}\qquad n=0,1,2,\cdots \tag{2.1}$$

式中 $z_0=(0,0)$，c_{pq} 为计算机荧屏位于 (p,q) 位置的像素。于是式（2.1）成了

$$z_{n+1}=(\cdots((((z_0^2+c_{pq})^2+c_{pq})^2+c_{pq})^2+c_{pq})^2+\cdots)^2+c_{pq} \tag{2.2}$$

给定 N 为一个正整数，比如等于 255，即画 256 色彩图。当像素位于 (p,q) 且 $n>N$ 时，复数 z_n 的模数 $|z_n|$ 仍然小于预设的一个阈值 K，则在 (p,q) 位置着色为 1（表示画蓝色，该区域就是著名的曼德勃罗湖或曼德勃罗港湾，观看彩色图片 1 之右图），否则当 $n\leqslant N$ 时，已有 $|z_n|\geqslant K$，则在 (p,q) 位置描色为 n（即 n 数字表示的颜色）。如此 (p,q) 遍历整个荧屏后，便画出了一幅彩色曼德勃罗集。该集合的坐标如图 2.3 所示。

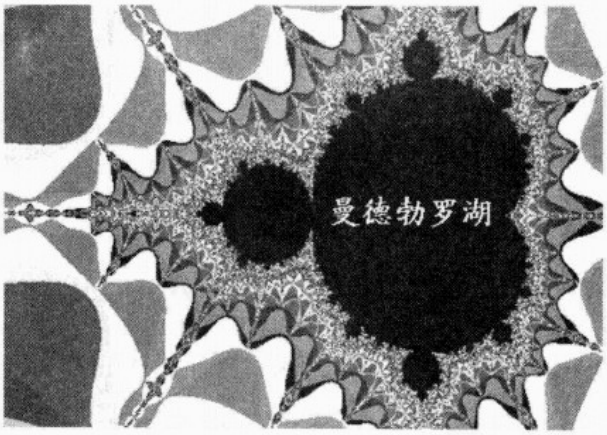


图 2.2　曼德勃罗集

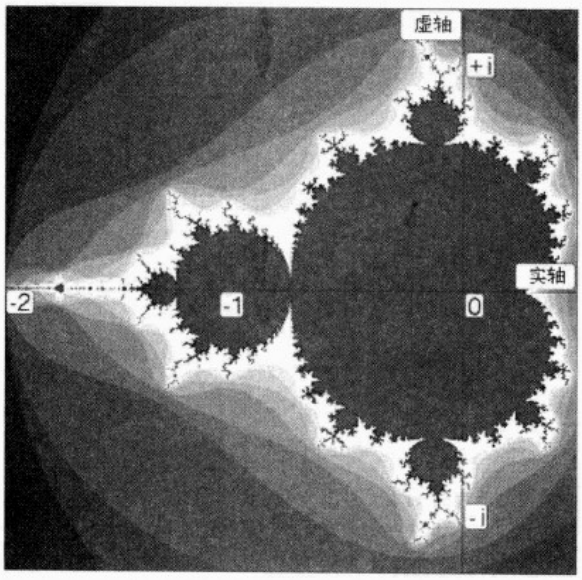


图 2.3　曼德勃罗集的坐标图

现在我们看到的曼德勃罗集具有多姿荆棘的圆盘，弯曲缠绕的螺线和细丝，挂着微细颗粒的鳞茎，无尽的斑斓色彩，意料外的精细结构，处处显示出分形奇特之美。20 世纪 80 年代后，曼德勃罗集被人们作为分形学和混沌学的国际标志出现在许多国际性论文集和刊物的封面上，成了世界巡回展出的计算机艺术精品。实际上，曼德勃罗集和同样震惊世界的茹利亚（Julia）集（见第 3 章）仅仅是映射 $z \to z^2 + c(z, c \in C)$ 的无穷次迭代。这种由数学的内在美变成人们视觉上的美，简直是匪夷所思。

为什么说曼德勃罗集是分形呢？实在是它的层层嵌套中有很多很多的自相似部分。图 2.4 显示了图中部分经逐级放大后，又出现了一个小的曼德勃罗湖。（从左到右和从上到下，后一个图就是前一个小方块的放大）。在我们附带的光盘中，有更多的显现小曼德勃罗湖成串出现的图片，其次曼德勃罗湖的湖岸线的几何维数不是整数。图 2.4 的绘画程序见程序 2.4。

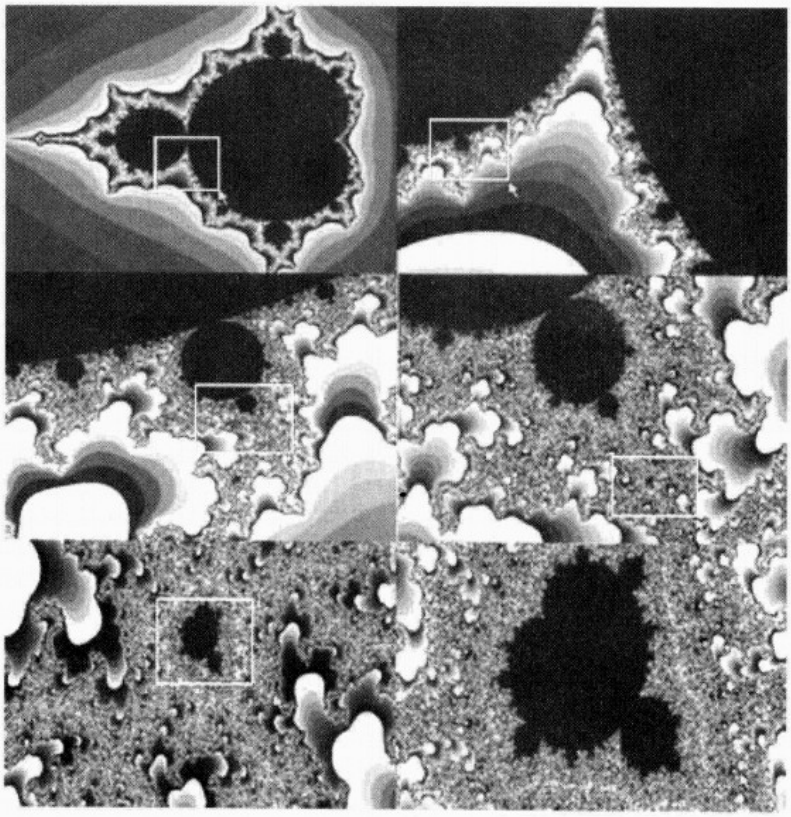


图 2.4　曼德勃罗集内嵌的小曼德勃罗集

一般来说，人们把图 2.2 所示的中间部分称为曼德勃罗湖，湖的四周有多处港湾。那么它的湖岸线是一条什么样的曲线呢？它的几何维数是多少呢？一直以来，人们都认为它是类似柯克（Koch）曲线一样的东西。但经更细心的观察后，发现它绝对不是如此简单。将湖岸线的局部经累次放大后，发现它的细节实在是太丰富了，见图 2.5。

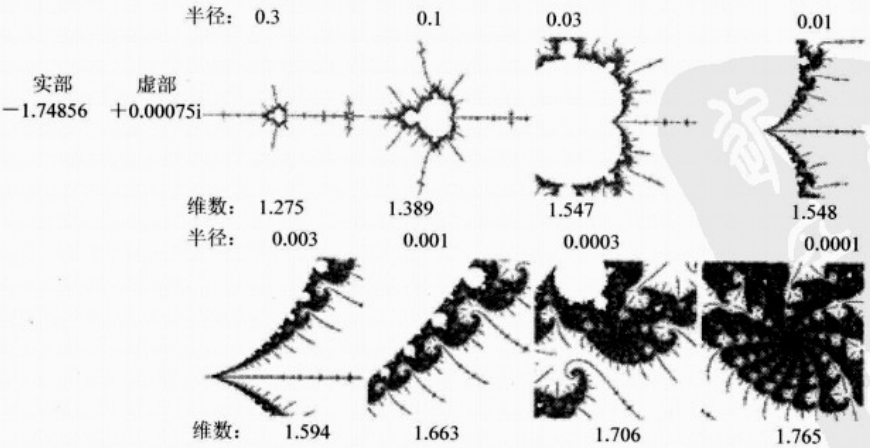


图 2.5　曼德勃罗湖岸上某点不同半径范围内的图形维数

定义 2.2　若对某些$n\in\{1,2,3,\cdots\}$，存在$z\in C$使得$f^n(z)=z$，则称点$z\in C$为f的周期点，这个n称为z的周期。周期n的最小值称为周期点z的最小周期，f的一个周期点的轨道称为f的循环，f循环的周期是循环中点的周期。

在每个n循环（即n周期点的轨道）的元素（点）中，一定有一个值c使得$z_0=(0,0)$属于n循环，这种使$f^n(o)=0$的c点称为n循环的中心。

例 2.3　令$c=0$，则$f^1(o)=0^2+0=0$，即原点o是 1 周期点，也是 1 循环点。

例 2.4　$c=-1,z_1=0^2-1=-1,z_2=(-1)^2-1=0,z_3=-1,z_4=0,\cdots$所以$c=-1$是 2 循环元素点，

例 2.5　$c=-2,z_0=0,z_1=0^2-2=-2,z_2=(-2)^2-2=2,z_3=z_4=\cdots=2$。$z=2$称为不动点，在曼德勃罗集里，则称米休雷维茨（Misiurewicz）点。

所有这种米休雷维茨点都属于曼德勃罗集的边界且都在边界曲线分枝的尖端点、螺旋线的中心或曲线分枝的连接点。这些米休雷维茨点通过曼德勃罗集（以后简称 M-集）的边界散射开来。中心点在边界的每个循环都无穷尽地包围着许多米休雷维茨点。围绕 M-集边界点的每个循环都包围着 M-集的一个复制品（小 M-集）（见图 2.4），华裔学者谭蕾证明了围绕这些小 M-集的曲线交点序列c_n（米休雷维茨点），当$n\to\infty$时都有其维数趋于 2 的区域（见图 2.5，图中的维数D_{dh}见定义 2.3）。这正是 M-集边界线的复杂性体现。

定义 2.3　记$pop(x)$是x规模（尺寸）区域中集合的像素点总数，则δ豪斯道夫（Hausdorff）维定义为

$$D_{dh}=\lim_{s\to 0}\log_2\frac{pop(2s)}{pop(s)} \tag{2.3}$$

程序 2.2　曼德勃罗集。

```
#include <stdio.h>
#include <graphics.h>
#include <complex.h>
#include <conio.h>

main ()
{
complex c,z;
int max_iterations=256,row,col,color,gd=DETECT,gmode;
double max_size=1000.0,XMax,XMin,YMax,YMin,deltaP,deltaQ;
initgraph (&gd,&gmode,"") ;
XMax=1.5;    XMin=-2.2;
YMax=1.5;    YMin=-2.2;
deltaP =  (XMax - XMin) /800;
deltaQ =  (YMax - YMin) /600;
for  (col=0; col<800; col++)
{
for (row=0; row<600; row++)
   {
     c = complex (XMin + col * deltaP,YMax - row * deltaQ) ;
     z =  (0,0) ;
     color = 0;
     while  ((color < max_iterations)  &&  (norm (z)  < max_size))
     {
```

```
            z = z*z + c;
            color++;
          }
          if (color >= max_iterations)
          color = 1;
          putpixel (col,row,color) ;
        }
        }
    getch () ;
    closegraph () ;
    }
```

由加拿大滑铁卢大学（University of Waterloo）研制的符号语言软件“Maple”（V～XI 版），有很强大的绘图功能。用它来绘制曼德勃罗集，程序相对简单得多。下面是相应的程序和用它画的图（图 2.6）。

程序 2.3　曼德勃罗集（Maple 语言）。

```
# The Mandelbrot set
#
# 2D display coding by John Oprea, oprea@math.csuohio.edu

> restart: with (plots) :

> mandelbrot := proc (x, y)
>    local c, z, m;
>    c := evalf (x+y*I) ;
>    z := c;
>    for m from 0 to 30 while abs (z) < 2 do
>       z := z^2+c
>    od;
>    m
> end:

> plot3d (0, -2 .. 0.7, -1.2 .. 1.2, orientation=[-90,0], grid=[250,
250],
    style=patchnogrid, scaling=constrained, color=mandelbrot) ;
```

程序 2.4 是由 Java 语言编写的，它可以由鼠标的拖动让曼德勃罗分形的任意部位局部放大。图形效果见图 2.4，文中汉字是对程序的解释。

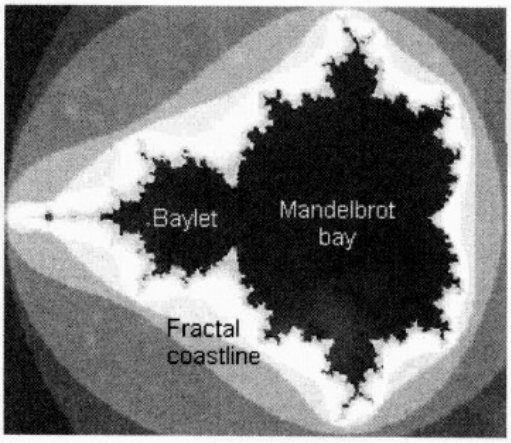


图 2.6　曼德勃罗集

程序 2.4　实现 Mandelbrot 集的绘制，并用鼠标来控制局部放大。

```
private int escapesToInfinity (double a, double b)
     {
        double x = 0.0;
        double y = 0.0;
        double d;
        int iterations = 0;

     do
        {
            double xnew = x*x -y*y + a;
            double ynew = 2*x*y + b;
            x = xnew;
            y = ynew;
            d = x*x+y*y;
            iterations++;
            if (iterations==MAX_ITERATIONS) return 0;
        }
        while (d<=20) ;
        return iterations;
   }//以上判断是否逃逸的函数
Color GetColor (int c)
   {
        float p = (float) c;
        float q = 1-p/50;
        Color color =new Color (q,q,1.0f) ;
        return color;
   }// GetColor 函数按逃逸时的迭代次数生成颜色
addMouseListener (new MouseAdapter ()
   {
        public void mousePressed (MouseEvent e)
        {
            x1 = -1;
            y1 = -1;
            x0 = e.getX () ;
            y0 = e.getY () ;
            Graphics g = getGraphics () ;
        }
        public void mouseReleased (MouseEvent e)
        {
            Graphics g = getGraphics () ;
            xmin = xmin+dx*x0;
            xmax = xmin+dx*x1;
            ymin = ymin+dy*y0;
            ymax = ymin+dy*y1;
            update (g) ;
        }
   }) ;
```

以上代码实现鼠标控制局部放大的范围，在选择放大范围时，我们需用到橡皮带技术，下面介绍一下橡皮带技术。

橡皮带技术是 applet 开发者工具集中最基本的工具之一，它在许多环境中都非常有用。例如：一个用于同时选定多个项目的方法，将在所有被选中项目周围绘制一个矩形橡皮带线。绘图应用程序在实际进行屏幕上的具体绘制操作之前，一般首先使用橡皮带技术让用户确定要进行绘制的图形。

橡皮带操作主要用于动态更新几何形状的边界：由于橡皮带操作通常在用户按下鼠标键时开始，拖动鼠标时继续，故在操作过程中，屏幕上将出现一个几何形状跟踪鼠标的运动，即几何形状边界是随用户的鼠标拖动操作而实时更新的。其中，几何形状的一角由鼠标开始点击的点定义，而共对角则由鼠标指针的位置定义，而且这个对角随鼠标移动而改变，用户在屏幕上可以看到几何形状随鼠标的移动而不断伸缩，像橡皮带一样，这也正是橡皮带技术的命名由来。

当然，利用 AWT 构件很容易完成几何形状的绘制。橡皮带技术的难点主要在于绘制新几何形状时要擦除原有形状。另外，在使用橡皮带的过程中还必须特别小心，切记新橡皮带不能干扰位于其下方的已有图形，即它应采用 XOR 模式绘制.*/

```
public void drawMandelbrot (Graphics g)
    {
        int color = 0;
        dx = (xmax-xmin) /WIDTH;
        dy = (ymax-ymin) /HEIGHT;
        for (int col=0; col<WIDTH; col++)
        {
            x = xmin+dx*col;
            for (int row=0; row<HEIGHT; row++)
            {
                y = ymin+dy*row;
                color = escapesToInfinity (x,y) ;
                if (color!=0)
                {
                    g.setColor (this.GetColor (color)) ;
                    g.drawOval (col,row,1,1) ;
                }
            }
        }
    }//上面是橡皮带技术的实现代码。
//最后，是画 Mandelbrot 集的主函数.
addMouseMotionListener (new MouseMotionAdapter ()
    {
        public void mouseDragged (MouseEvent e)
        {
            Graphics g = getGraphics () ;
            g.setXORMode (getBackground ()) ;
            if (x1!=-1&&y1!=-1)
                g.drawRect (x0,y0,Math.abs (x0-x1) ,Math.abs (y0-y1)) ;
            x1 = e.getX () ;
            y1 = e.getY () ;
            g.drawRect (x0,y0,Math.abs (x0-x1) ,Math.abs (y0-y1)) ;
        }
}) ;//至此，就完成了 Mandelbrot 集的绘制
```

2.3　曼德勃罗分形结构的复杂性和艺术性

曼德勃罗分形的结构非常复杂，为了叙述这一点，我们在曼德勃罗分形里找出六个小块，它们分别是顶部（top）、侧边点邻域（side）、半岛（peninsula）、边境点邻域（end）、半岛顶点邻域（point）和折叠点邻域（fold）等，见图 2.7。这六个小块大小不一，有的较大，如顶部。有的很小，如侧边点邻域、半岛顶点邻域等。

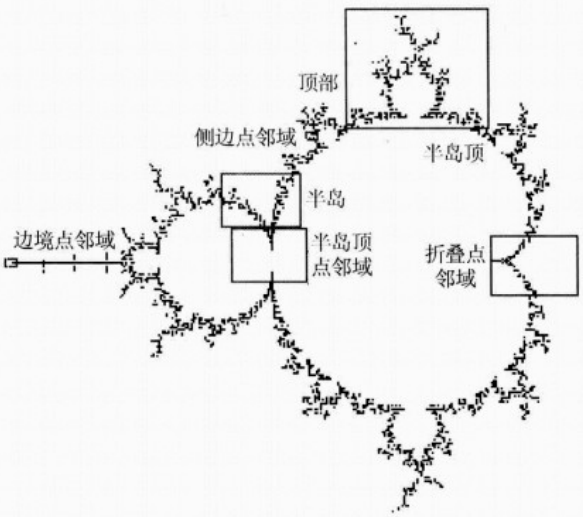
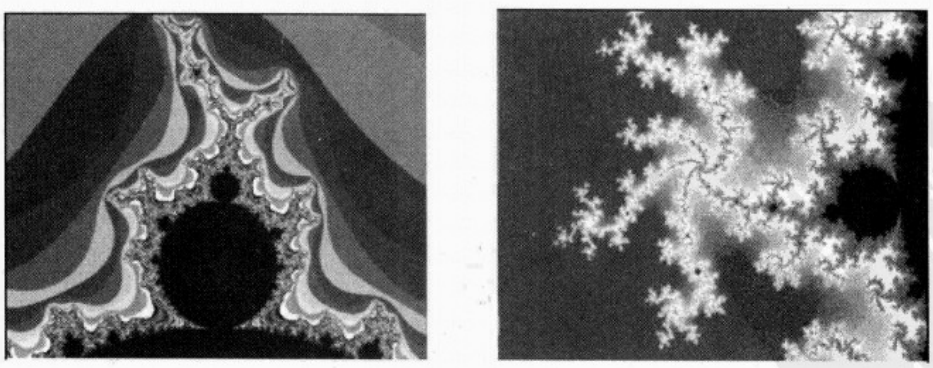


图 2.7　曼德勃罗分形的六个小块

下面对每个小块做较为详细的分析。

2.3.1　顶部

顶部的图形如图 2.8（a）所示。



（a）　　（b）

图 2.8　曼德勃罗分形的顶部（a）与侧边点邻域（b）

顶部的一小块放大后呈现出绚丽多彩的龙卷风画面，见图 2.9。图的右面小图显示该龙卷风在顶部的位置。下面的同类插图中也表示同一个意思，以后不再赘述。

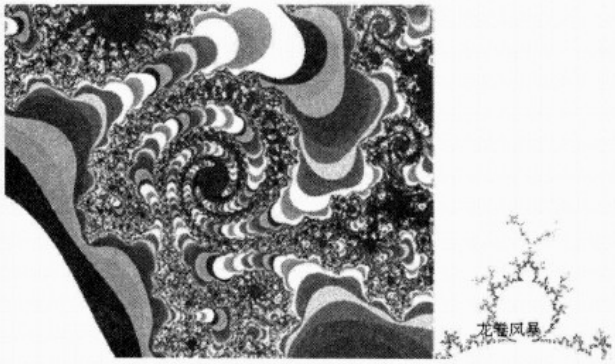


图 2.9　顶部位置的龙卷风暴

2.3.2　侧边点邻域

侧边点邻域的放大图见图 2.8（b）。看得出，在侧边点域出现了若干小曼德勃罗湖。在该点域的某部位，经放大后出现了曲折的蜿蜒道，见图 2.10。

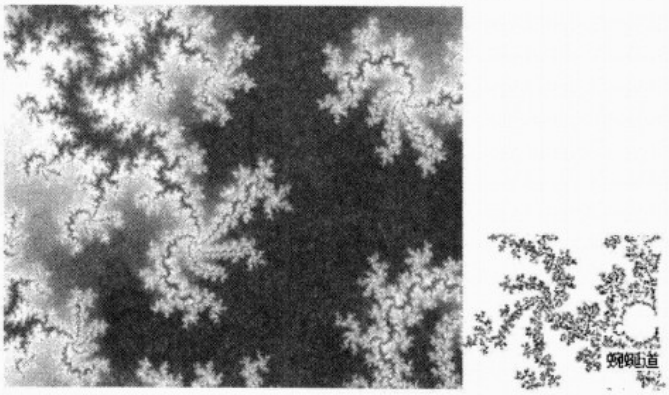
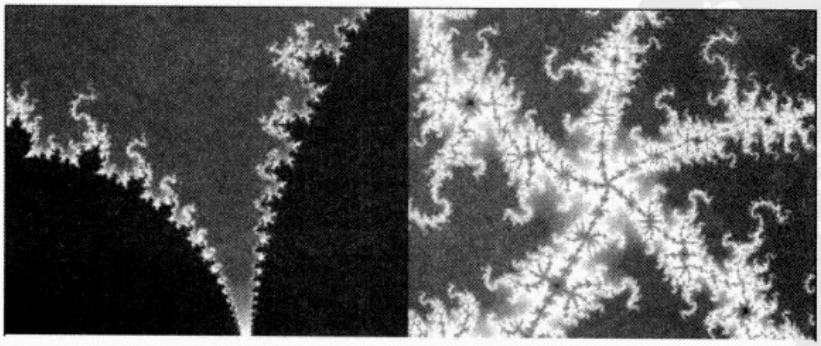


图 2.10　侧边点邻域的蜿蜒道

2.3.3　曼德勃罗半岛

半岛图见图 2.11（a）。放大的海星见图 2.11（b）。



（a）　　（b）

图 2.11　曼德勃罗半岛（a）与半岛上的海星（b）

半岛区域的地形更具复杂性。当中有海星、螺旋器、地狱、水流和扶梯等，分别如图 2.12～图 2.15 所示。

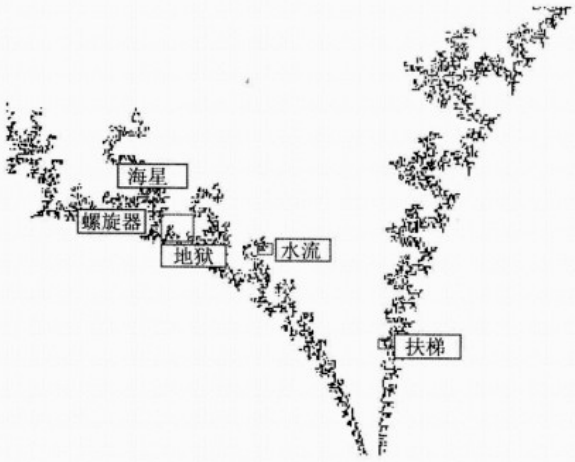


图 2.12　半岛中的海星、螺旋器、地狱、水流和扶梯

无独有偶，这里仍然出现数个曼德勃罗湖。

螺旋器见图 2.13（a），地狱见图 2.13（b）。螺旋器附近也出现了曼德勃罗湖。它的一个部位放大后呈皂质状，见图 2.14。

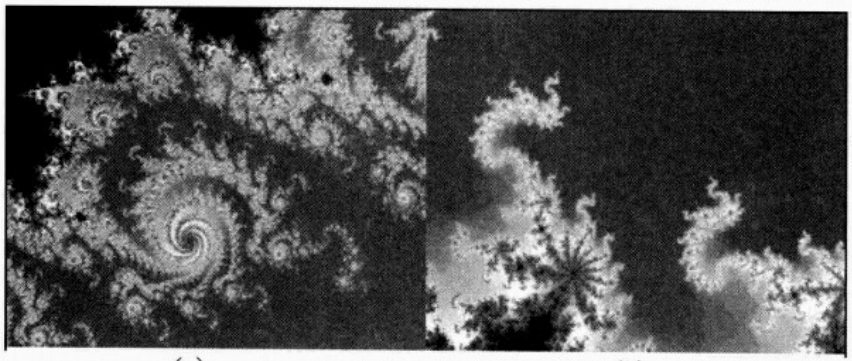


图 2.13　半岛中的螺旋器（a）与地狱（b）

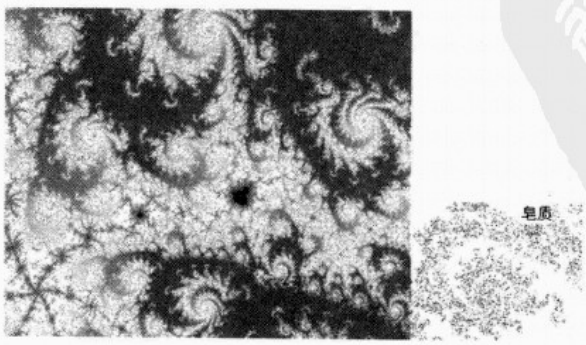
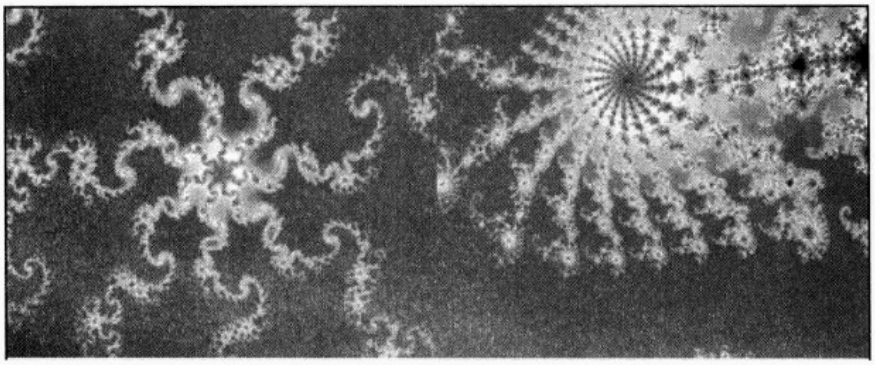


图 2.14　螺旋器中的皂质状



(a)　　(b)

图 2.15　半岛上流水四溢（a）与众多的扶梯（b）

吃惊的是，在图 2.15（b）的扶梯上竟然也挂着曼德勃罗湖。

2.3.4　边境点邻域

边境点邻域见图 2.16。

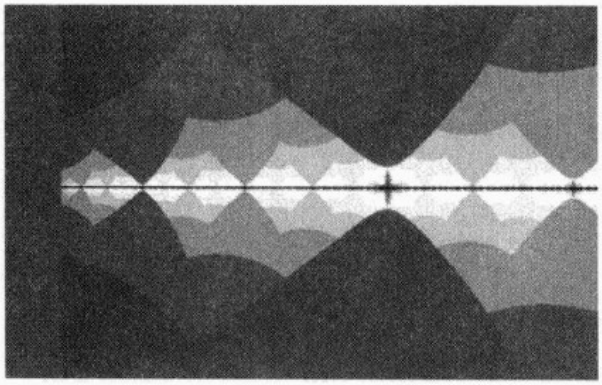


图 2.16　边境点邻域

在边境地区居然有光芒四射的地方，见图 2.17。

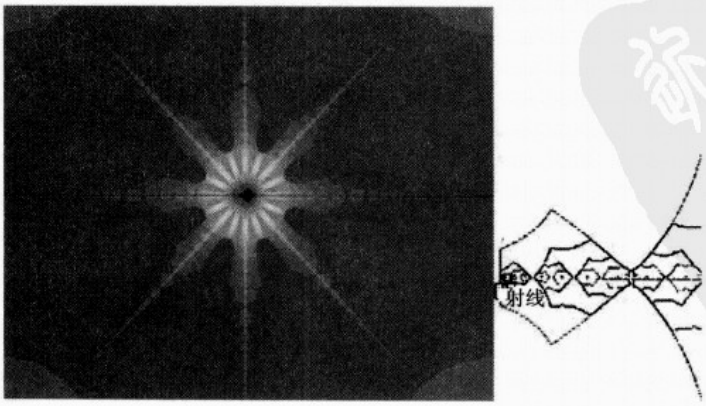


图 2.17　边境地区的流光射线

在发光的中心点自然是“太阳”了，仔细看看，且原来又是曼德勃罗集，如图 2.18 所示。

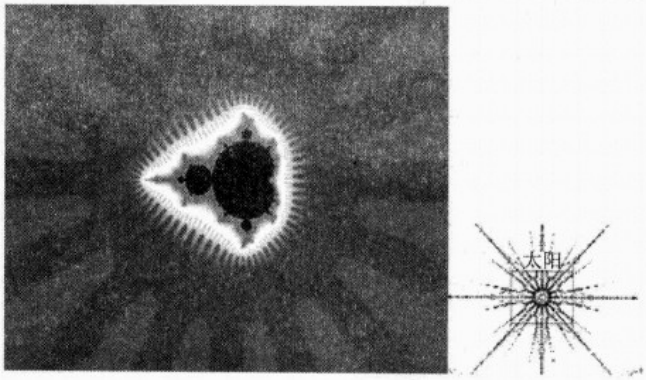


图 2.18　边境地区闪光的曼德勃罗太阳

在曼德勃罗集的边缘地带，我们又再次见到了更小的曼德勃罗集，如图 2.19 所示。

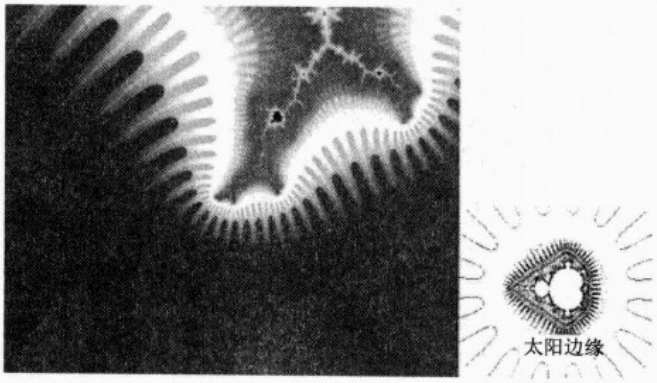


图 2.19　“太阳”边缘地带的曼德勃罗集

2.3.5　半岛顶点邻域

在半岛顶点邻域，难道还有什么奥妙吗？先看看半岛顶点部分的放大情况，见图 2.20（a），并且在最尖端区域居然有“漂亮”的陷阱存在，见图 2.21。

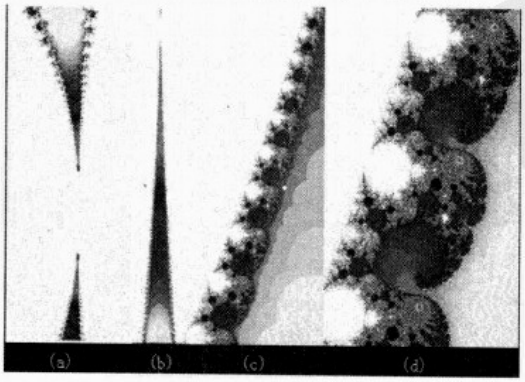


图 2.20　半岛的尖锥点（a）和折叠区域的彗星（b）串珠（c）和羽状物（d）

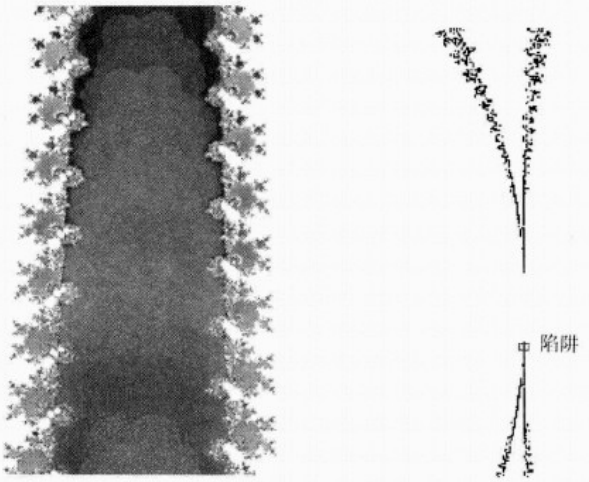


图 2.21　半岛尖锥点里的陷阱

漂亮的陷阱边缘，有好看的大小鱼眼一对，见图 2.22。

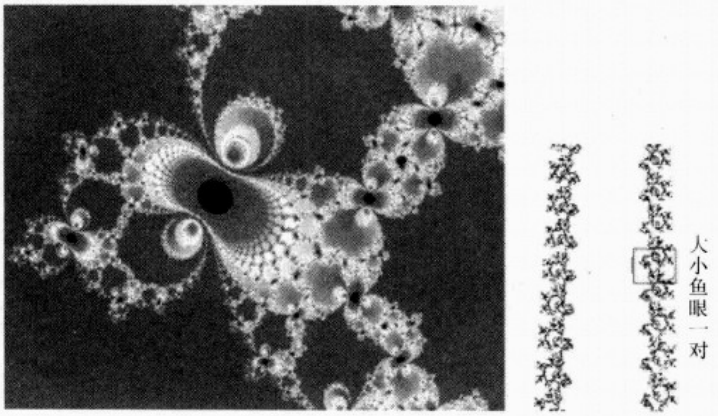


图 2.22　陷阱边缘的大小鱼眼一对

这一大一小鱼眼睛，放大以后，小眼睛变成了一双对眼，大眼睛原来是一个大眼球，见图 2.23。

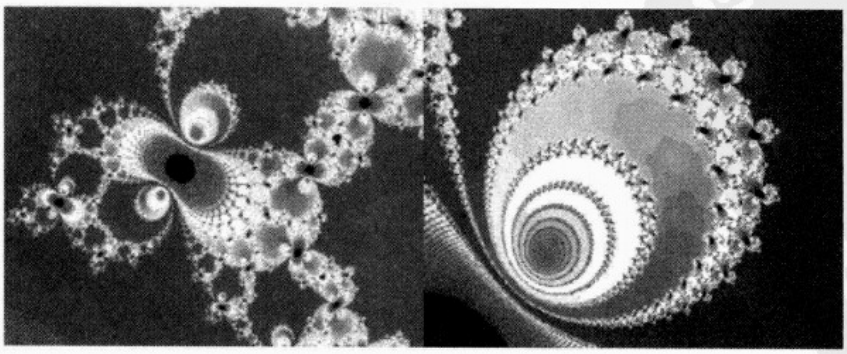


图 2.23　对眼和大眼球

2.3.6　折叠点邻域

最后，了解一下折叠点邻域的结构情况。我们截取该地区且把它们放大后，得到图 2.24。

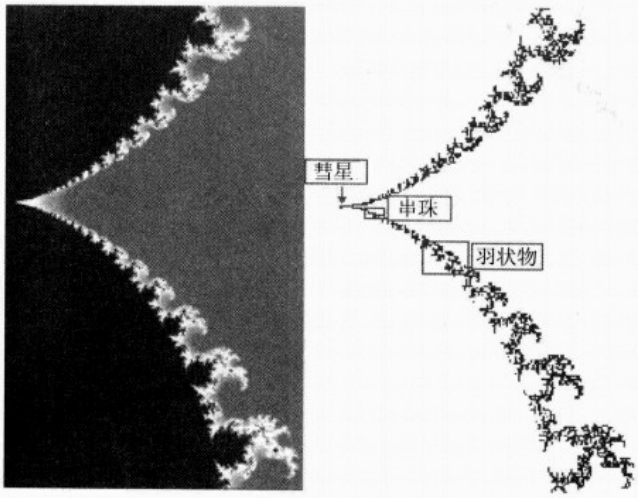


图 2.24　折叠点邻域

图 2.24 位置就是图 2.20 中（b）、（c）、（d）三个图。从左到右依次为折叠点邻域中的彗星、串珠和羽状物。

在彗星中取出小涓滴（droplet），居然也煞是好看，见图 2.25。

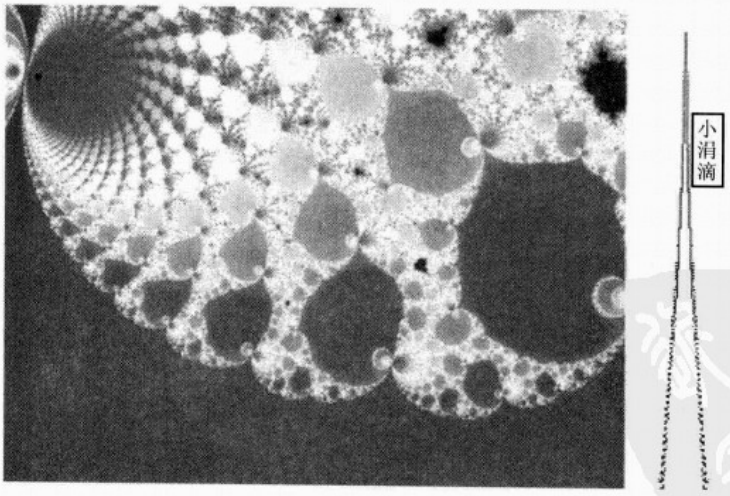


图 2.25　彗星中的涓滴

在串珠中摘取两颗，看其庐山真面目，原来是涡流状的东西，见图 2.26。

从一颗珠宝里截取一小块放大，恰如精细的网织品，当中又有两个曼德勃罗湖，见图 2.27。

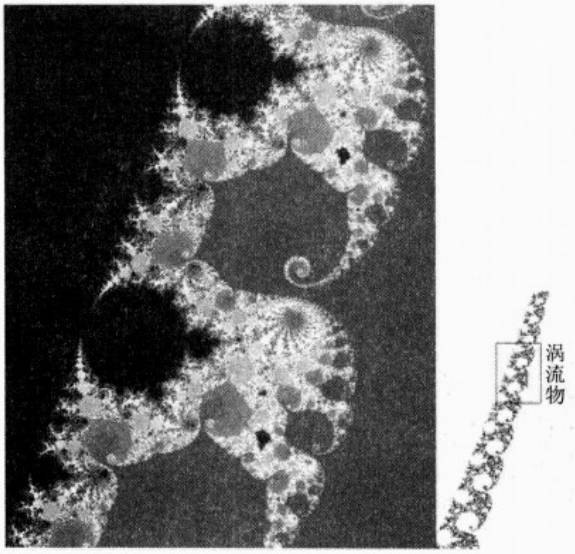


图 2.26　串珠中的两颗涡流物

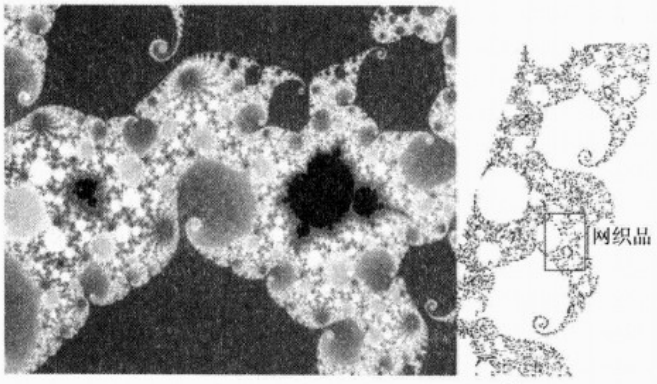


图 2.27　精细的网织品

从图 2.24 的羽状物中，截取一小片，像是爪钩的东西，见图 2.28。

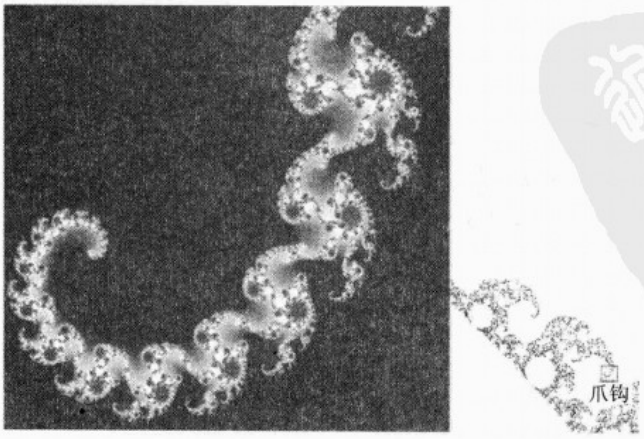


图 2.28　羽状物中的爪钩

从爪钩中又可截取到一架“直升飞机”，见图 2.29。

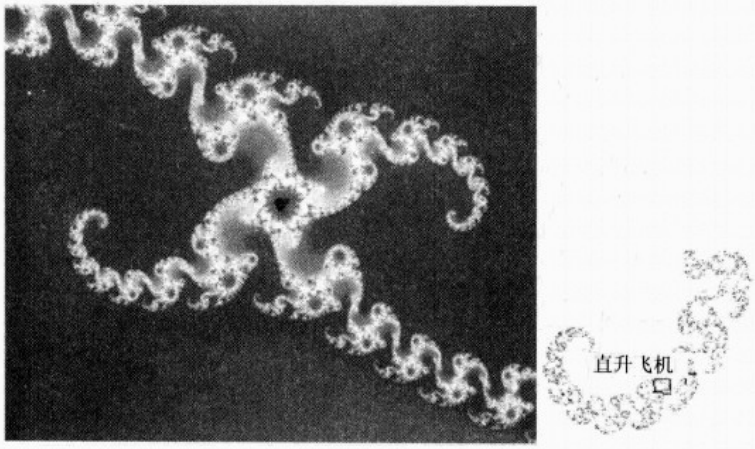


图 2.29　爪钩里飞出来的直升飞机

直升飞机顶上的螺旋桨是什么样的？看看下面图 2.30，可知其中心原来又是个曼德勃罗集。

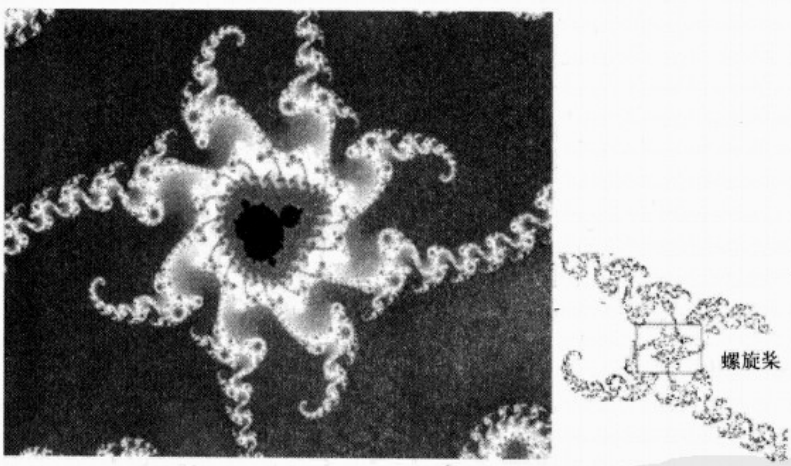


图 2.30　中心是曼德勃罗集的螺旋桨

对上面六个部位的剖析中，我们不仅得到各种各样五彩缤纷的 M-集图案，还一再看到了小 M-集的出现。这正是 M-集的奇妙之处之一。

图 3.31 从曼德勃罗集全图出发再次显示了图中某点区域经放大后得到的漂亮图案。

上面我们一再看到 M-集局部区域的放大，而这种累次的放大在理论上应该是无穷尽的。但它们被放大以后，它们在原图的坐标系（见图 2.3）中处于一个什么样的位置呢？图 2.32～图 2.33 显示了这一点。图中下部左右两个数字分别表示实部的下界和上界，左面的上下两个数字则是虚部的上界和下界。

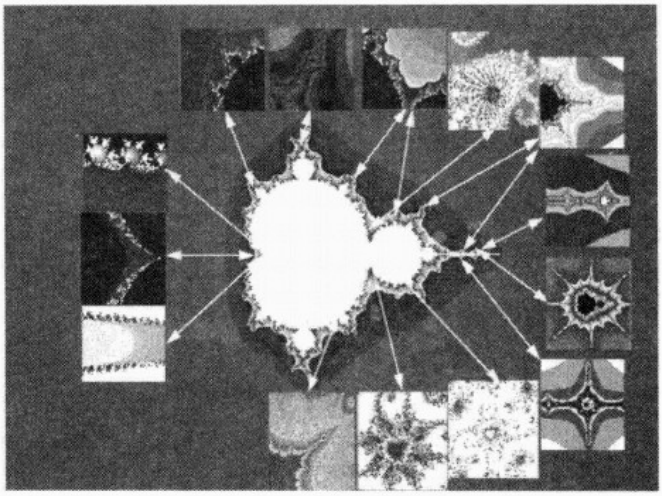


图 2.31　曼德勃罗集里某点域及其放大的图

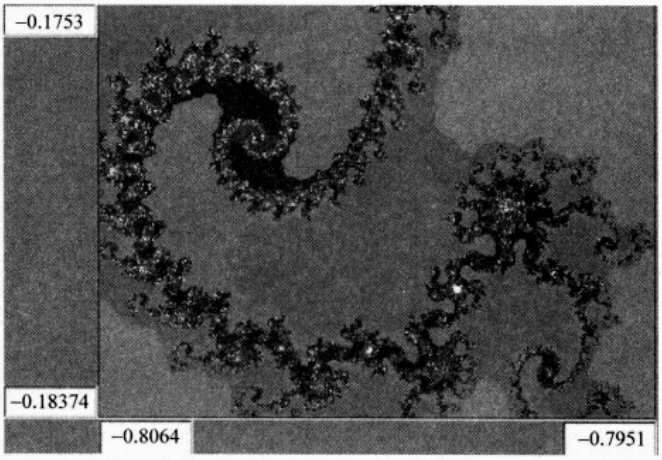


图 2.32　局部图形放大后的坐标之一

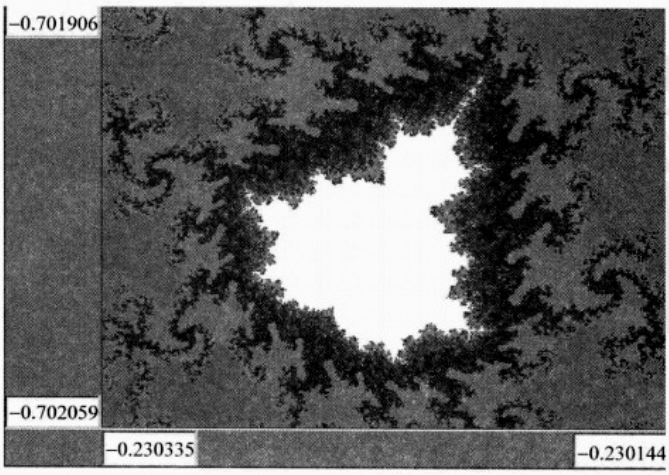


图 2.33　局部图形放大后的坐标之二

从图 2.33 来看，实部和虚部区域的长度分别为 0.000191 和 0.000153，这在图 2.3 中，仅表示 0.03×0.024 个像素大小。现在这张图，垂直上下与水平左右有 386×294 个像素，这说明该图在图 2.3 的位置上被放大了 12867×12250=157620750 倍。

2.4　广义曼德勃罗分形

经典的 M-集是由映射 z^2+c 得来，人们自然会对该映射采用更多的变化规律和更多的其他映射，从而得到各种各样的 M-集，我们称它们为广义 M-集。

2.4.1　从变化映射 z^2+c 的幂次而得到的广义曼德勃罗集

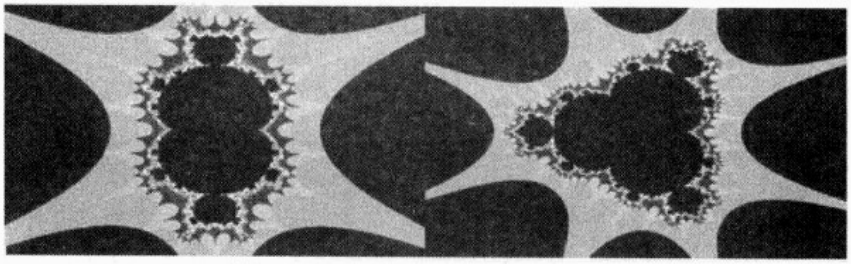
这种更换 z^2+c 的映射规律是无穷尽的，这里略举几个。

1）整数次幂广义 M-集合。迭代公式为

$$z_{n+1}=z_n^m+c \qquad z_n, c\in C,\ n, m \text{ 为整数} \tag{2.4}$$

$m=2$ 时，就是前述的 M-集。

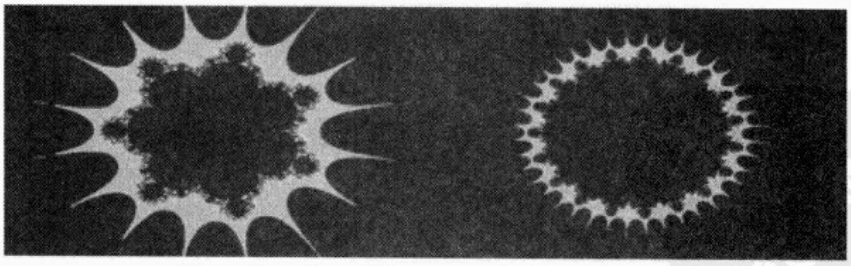
$m=3$ 和 $m=4$ 时，图形见图 2.34 的（a）和（b）。



（a）　　（b）

图 2.34　三次和四次 M-集

$m=8$ 和 $m=20$ 时，见图 2.35 的（a）和（b）。



（a）　　（b）

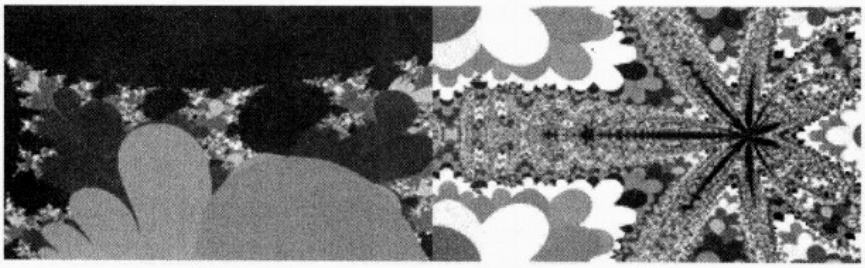
图 2.35　八次和二十次 M-集

可以发现这样一个规律，m 次的 M-集合有 $m-1$ 个大港湾。且当 m 愈来愈大时，其图形将愈来愈接近一个圆。

2）复数次幂广义 M-集。迭代公式为

$$z_{n+1}=z_n^p+c, z_n, p, c\in C, n \text{ 为整数} \tag{2.5}$$

令 $p=(2.71828, 3.14159)$ 时，见图 2.36（a）。



（a）　　（b）

图 2.36　指数为（e,π）的 M-集（a）和二次幂带修正项的 M-集（b）

3）二次幂带修正项的广义 M-集。迭代公式为

$$\left.\begin{aligned} z_{n+1} &= z_n^2 + y_n + c \\ y_{n+1} &= z_n \end{aligned}\right\}, \quad z_n, y_n, c \in C \tag{2.6}$$

产生的图见图 2.36（b）。

4）三次幂带函数项的广义 M-集。迭代公式为

$$z_{n+1} = z_n^3 + z_n + p\sinh c \quad z_n,\ p, c \in C \tag{2.7}$$

令 $p=$（0.26, −0.29），得到图 2.37。

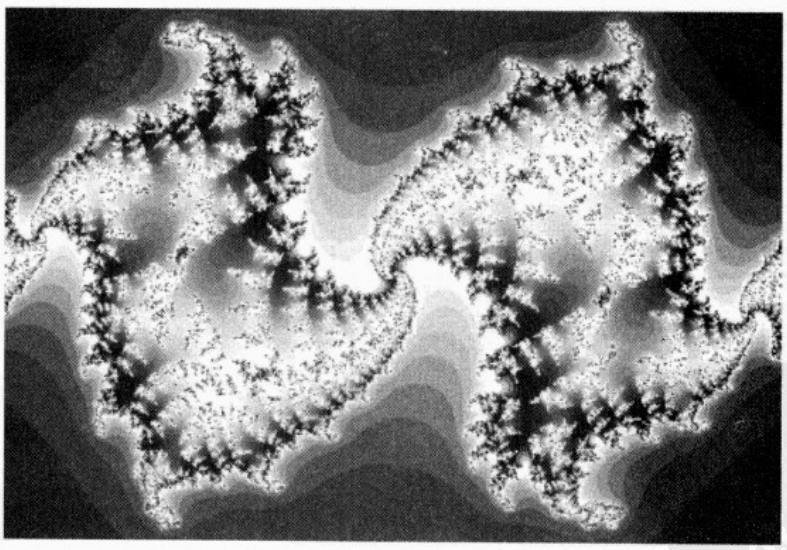


图 2.37　三次幂带函数项的广义 M-集

其实这类广义 M-集是无穷无尽的，上面仅是极少数的几个。

2.4.2　其他函数代替 z^2 的广义曼德勃罗集

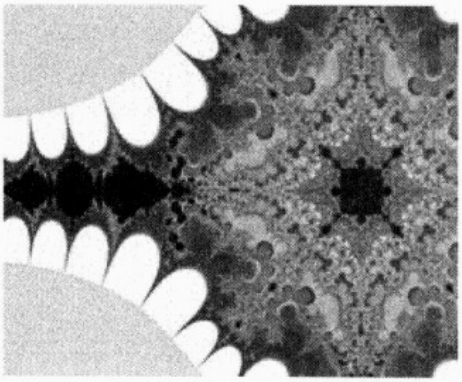
虽然基本初等函数只有几个，但产生的初等函数却是数不胜数。这里设计了一种迭代格式，挑选了几个简单的函数，看看其图形会是怎样的。

1）迭代公式是

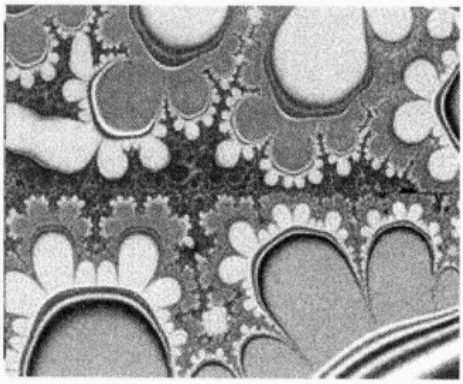
$$z_{n+1} = f_1^2(z_n) + f_2(c) \tag{2.8}$$

令 $f_1(z) = \sin z^2$，$f_2(c) = \cos c^2$，得图 2.38（a）。

令 $f_1(z) = \cos z^2$，$f_2(c) = \arccos c^2$，得图 2.38（b）



（a）

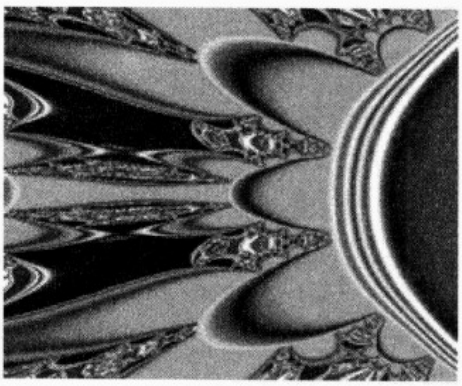


（b）

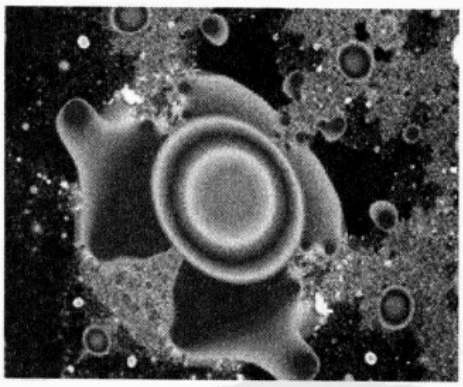
图 2.38　sin（z*z）平方 M-集和 cos（z*z）平方 M-集

令 $f_1(z) = \cosh z$，$f_2(c) = \sin c$。得图 2.39（a）。

令 $f_1(z) = \mathrm{c}\tan z$，$f_2(c) = \arcsin c$，得到图 2.39（b）。



（a）



（b）

图 2.39　cosh(z) 平方 M-集和 *ctanz* 平方 M-集

2）另一种迭代公式是：如果 $|z_n| < M$，则

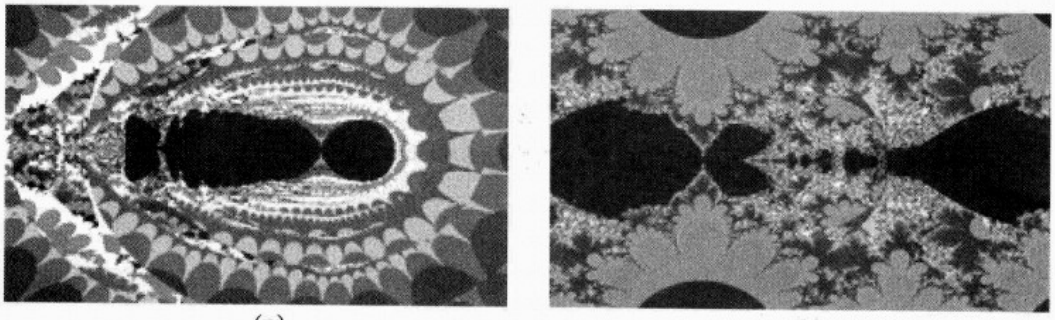
$$z_{n+1} = f_1(z_n) + c$$

否则

$$z_{n+1} = f_2(z_n) + c \tag{2.9}$$

令 $M = 1.0$，$f_1(z) = \sin z$，$f_2(z) = \sqrt{z}$。得到图 2.40（a）。

令 $M = 2.0$，$f_1(z) = e^z$，$f_2(z) = \cos z$，得图 2.40（b）。



(a)　　(b)

图 2.40　$\sin z$ 与 $\sqrt{z}$ 复合的 M-集和 e^z 与 $\cos z$ 复合的 M-集

彩色图片 1 是两张另一种广义 M-集，它是在原曼德勃罗集基础上经修饰处理后得到的。

2.5　伪三维曼德勃罗分形

自从出了曼德勃罗集这个计算机视觉艺术精品后，许多人又对其进行尽其所能的加工和渲染，让它成为三维（3D）作品，有立体的美感。

本章介绍的并不是真正的三维立体曼德勃罗集，真正的三维立体曼德勃罗集将在第 6 章的四元数分形中叙述。这里介绍的仅是 2D 曼德勃罗集的立体化，故称伪三维曼德勃罗集。

一种叫做拟 3D 分形图，即把二维图形置身于三维空间，毕竟这比纯粹的 2D 作品更好看一些。图 2.41 就属于这一类。我们把一个二维图像悬挂在海面上，图下是光线透过去的涟漪波浪。立体感虽不太强，但却有清新的感觉。图的左面是二维图像，右面是其加工品。图 2.42 的左半部则是图右部分的立体加工。

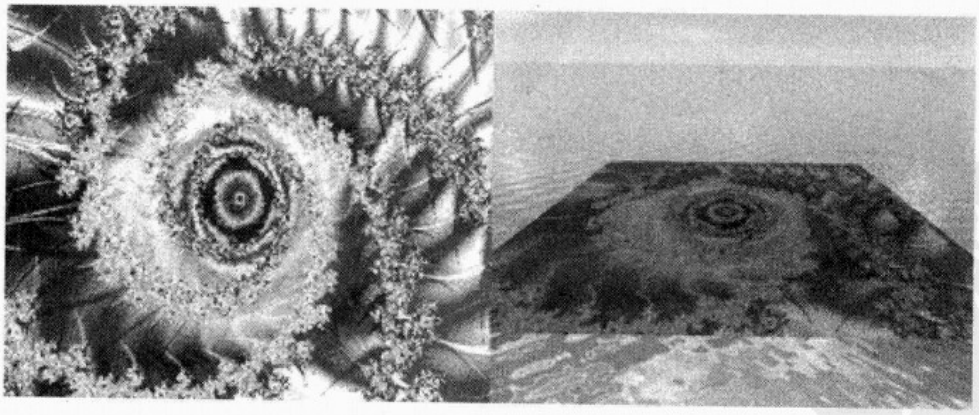


图 2.41　金色夕阳下海面上的曼德勃罗集

第二种是把原图像人为加厚。比如对原始图 2.2，我们可以把它加工为图 2.43（a）。图中把曼德勃罗湖置于峰顶，原湖岸线以外地区成了陡峭的壁岩。图 2.43（b）则是用 Maple 软件画的，在程序 2.3 中，只要把画图语句改成下面的 plot3d 就行。

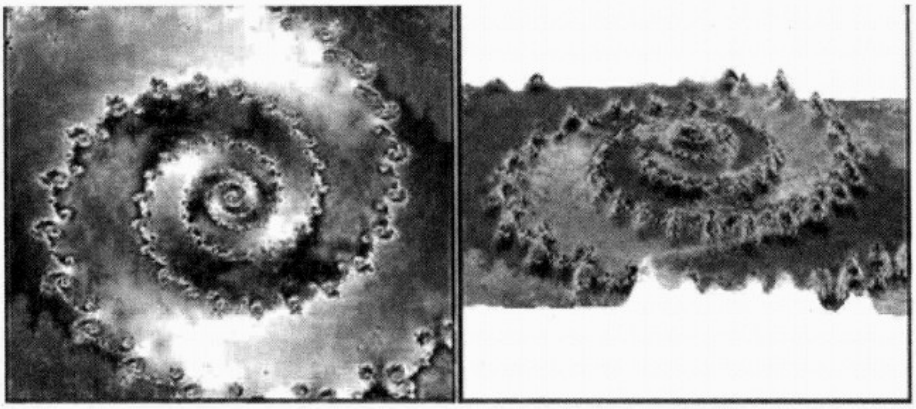
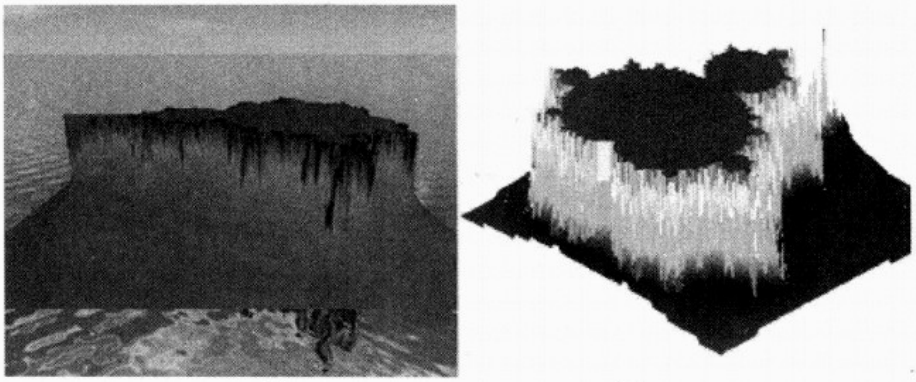


图 2.42　曼德勃罗集的立体化



(a)　　(b)

图 2.43　伪三维曼德勃罗集之一

```
> plot3d (mandelbrot, -2 .. 0.5, -1.2 .. 1.2, grid=[200, 200],
style=patchnogrid, shading=zhue, orientation=[45, 35]) ;
```

程序 2.5 可以把 2D 的曼德勃罗集转为伪 3D 图形。

程序 2.5　伪 3D 曼德勃罗集。

```
/***********************************************
*   Mand3DlandZ2.c
*
*  Generates a RAW format file of a 3D image
*  of Mandelbrot set  (grayscale)
*
***********************************************/

#include<stdio.h>

/* definition of constants */
#define Radius 10
#define Side 0.025
#define Cx 0.1620
#define Cy -1.035
#define M 700
#define Num 255
```

```
#define height 2*Num

/* main program */
main ()
{
    int p, q, n, w, r[M+height];
    double x, y, xx, yy, Incx, Incy;
    FILE *fp;
    fp = fopen ("Mand.bmp","w") ;

    for (p = 1; p <= M; p++)
    {
        printf ("%i %%\n", p*100/M) ;
    for (q = 1; q <= M; q++)
    {
        Incx = Cx - Side + 4*Side* (p+q/2) /M/3;
        Incy = Cy - Side + 2*Side*q/M;
    x = 0;
    y = 0;
    w = 100;
        for ( n = 1; n <= Num; ++n)
        {
            xx = x*x - y*y - Incx;
            yy = 2*x*y - Incy;
            x = xx;
            y = yy;
            if ( x*x + y*y > Radius )
                {
                    w = n;
                    n = Num;
                }
        }
                    r[q + height/w] = w;
    }
    for ( q = 1; q <= M; q++)
        {
    fprintf (fp, "%c", r[q] ) ;
    r[q] = 100;
        }
    }
    fclose (fp) ;
}
/* end of main program */
```

这种把曼德勃罗湖放在山顶的做法盛行一时，彩色图片 2 的下半部分的两幅图就是这种模式，左边一幅是局部地区的放大，旁边还配以闪电。上半部分的两幅图则是对曼德勃罗集或其局部的艺术加工。

也有把曼德勃罗集雕刻成凹进去的样子。图 2.44～图 2.45 就是如此。图中，左边的是二维原像，右边是对应的三维图。

虽然前面的几幅图（如彩色图片 2）颇有高山出平湖的感觉，但我们觉得还是湖的周围是山峦的为好。图 2.46 和图 2.47 就是这样画的。图 2.46 中，把船开进了曼德勃罗湖。图 2.47 中，在曼德勃罗湖周围的群山则是险峻的岩石高地，并且水从大湖哗哗流进了另一个港湾。

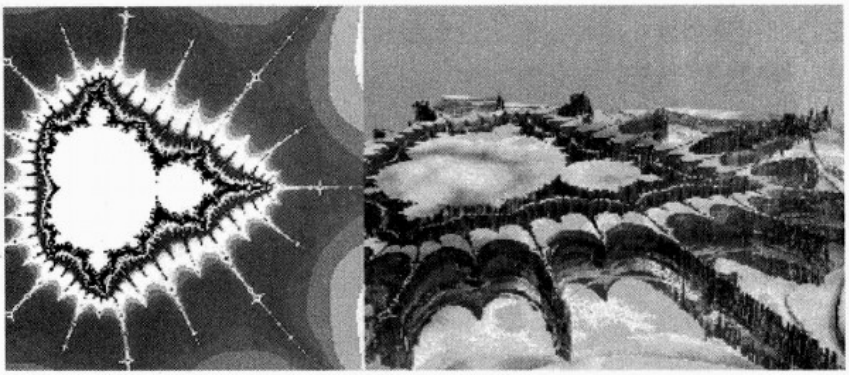


图 2.44　二三维对照图之一

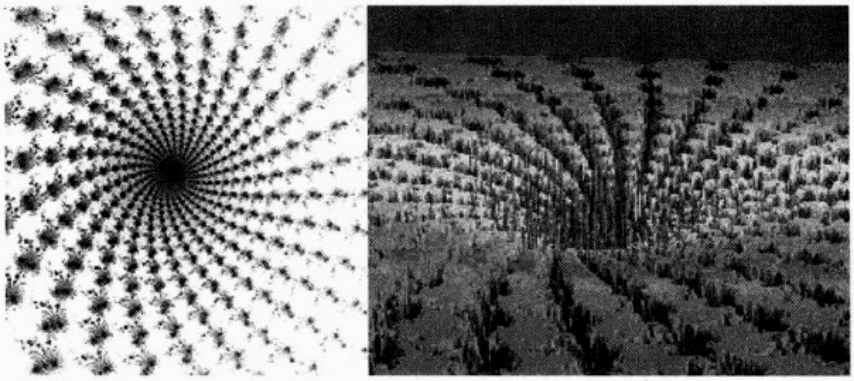


图 2.45　二三维对照图之二

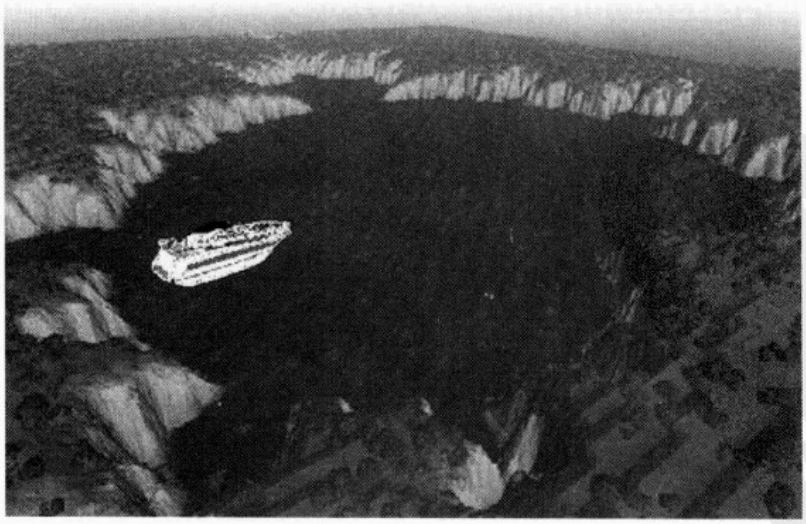


图 2.46　船行驶在曼德勃罗湖

有把曼德勃罗集画成更多形式的三维作品问世，见图 2.48～图 2.50。

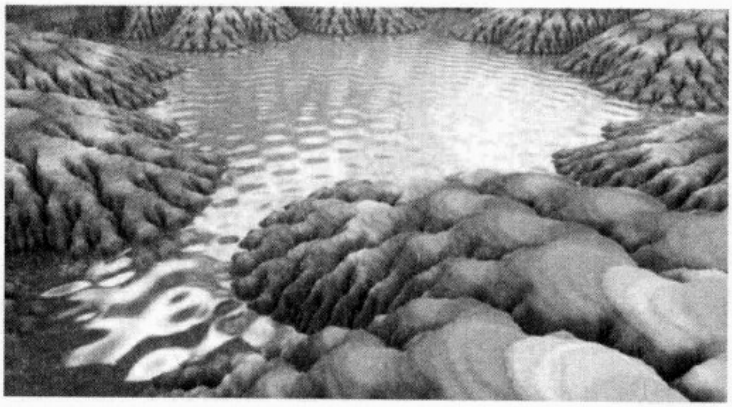


图 2.47　岩石高地包围着的曼德勃罗湖

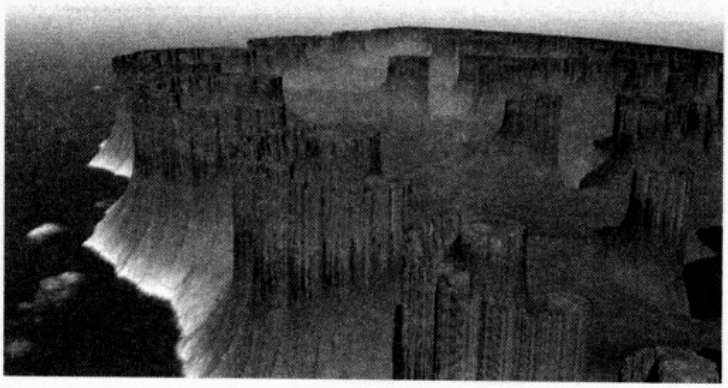


图 2.48　曼德勃罗集陡峭的湖岸

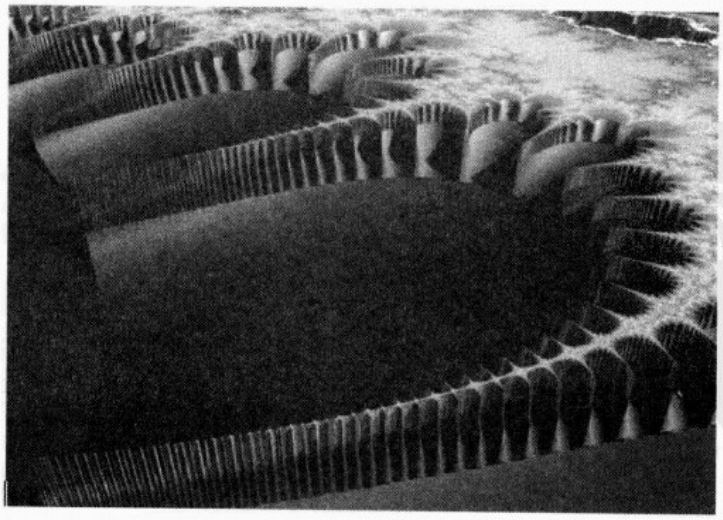


图 2.49　曼德勃罗湖曲折的港湾

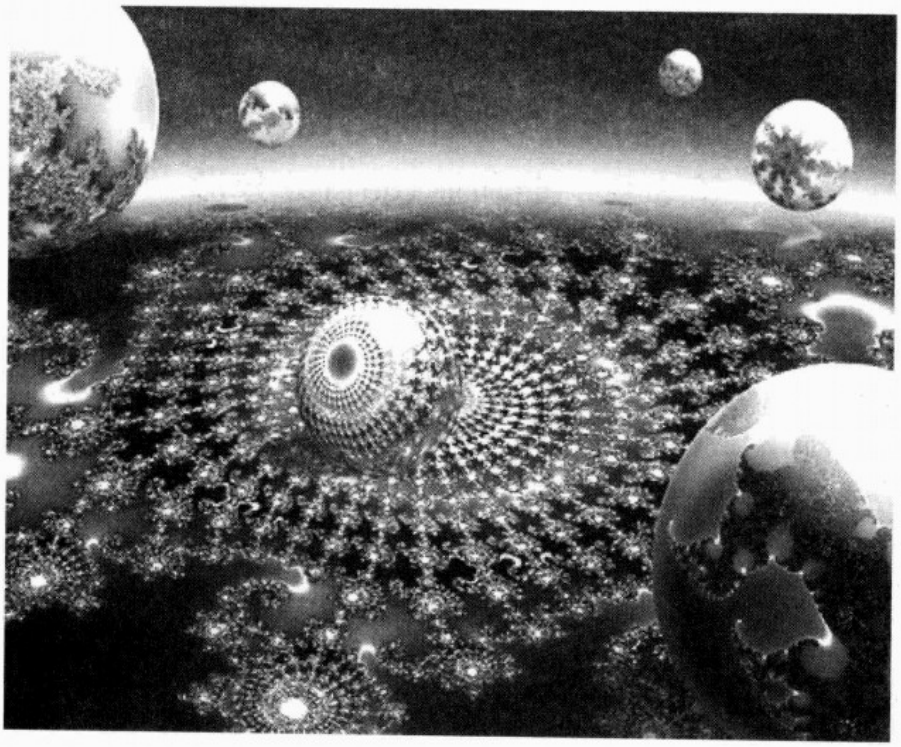


图 2.50　天上地下的曼德勃罗球

第 3 章　茹利亚(Julia)分形

我们从第 2 章迭代映射

$$z_{n+1}=z_n^2+c$$

出发描绘了著名的曼德勃罗分形。如果我们把复数 z_0 并不是固定在原点(0,0)，而是让它遍历整个荧屏，常数 c 却被固定，看看会画出一幅什么样的图形来。其实这个想法早在 1919 年第一次世界大战时就被法国数学家嘉斯顿·茹利亚(Gaston Julia，1893～1978)研究过了。那一年，他在战场上受了伤，住院期间他潜心研究了迭代保角变换 $z_{n+1}=z_n^2+c$。他认定这种复平面上的变换能生出一系列令人眼花缭乱的图形来。但当时没有电子计算机，不能像现在那样把如此美妙绝伦的图案奉献于世，因此他的工作并不为世人所重视。现在我们借助于电脑，很容易把茹利亚的美妙想法变成荧屏上的一幅幅精彩的图画。

3.1　茹利亚集的算法及其维数和性质

设 c 为复常数，z_0 为计算机荧屏上的某一个像素，把逃逸时间法用于映射

$$z_{n+1}=z_n^2+c \qquad n=0,1,2,3,\cdots \tag{3.1}$$

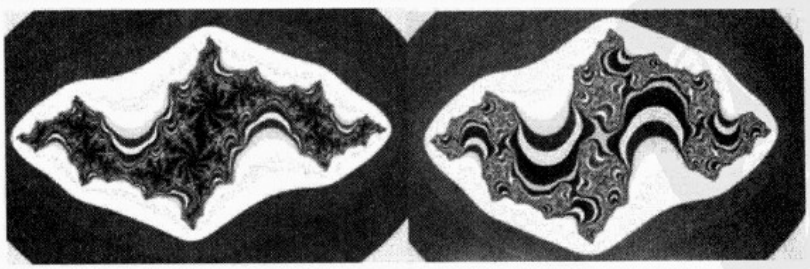
则当 z_0 遍历整个荧屏时，在荧光屏上画出的图形就称之为茹利亚集。

与公式（2.2）比较，这时的迭代变成了

$$z_{n+1}=(\cdots((((z_{pq}^2+c)^2+c)^2+c)^2+c)^2+\cdots)^2+c \tag{3.2}$$

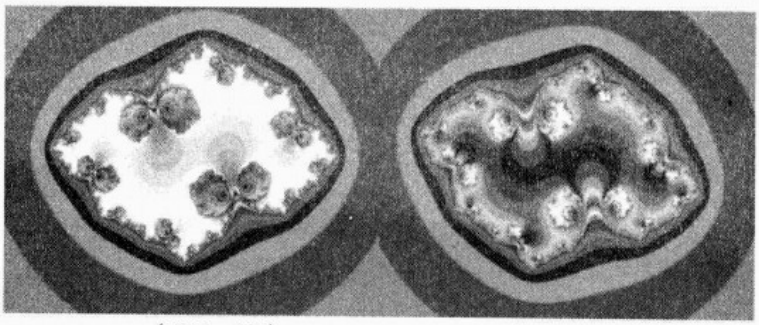
式中 z_{pq} 是位于位置 (p,q) 的像素复数值，c 是任意一个复常数。

图 3.1、图 3.2 和图 3.3 是分别对应于不同常数 c 的茹利亚集。



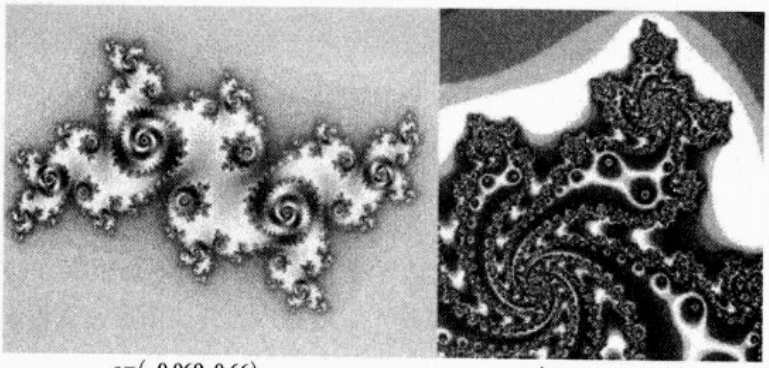
$c=(1.0,\ -0.1)$　　$c=(0.75,\ -0.13)$

图 3.1　茹利亚集之一



$c=(-0.29, -0.34)$　　$c=(-0.141, -0.45)$

图 3.2　茹利亚集之二



$c=(-0.069, 0.66)$　　$c=(-0.4701, 0.5664)$

图 3.3　茹利亚集之三

图 3.4 和图 3.5 是著名的另两幅茹利亚集。

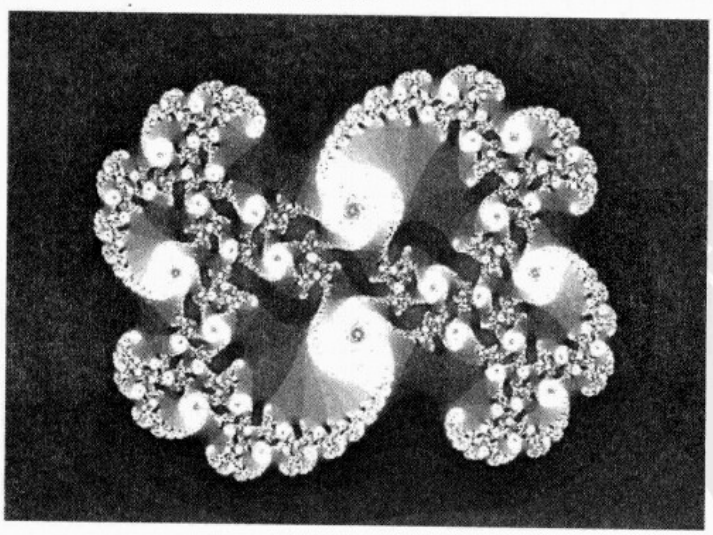


图 3.4　茹利亚集之四

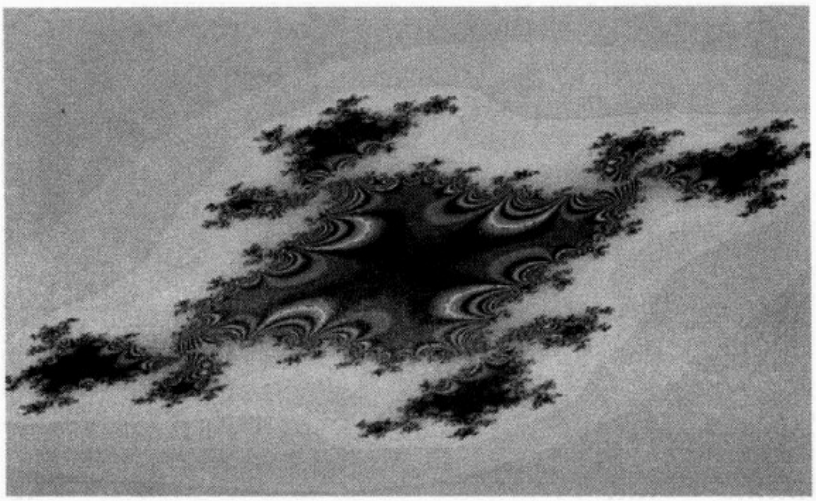


图 3.5　茹利亚集之五

彩色图片 3 和彩色图片 4 是四幅经艺术加工后的茹利亚集。

显然，对于每一个不同的复常数 c，就对应着不同的茹利亚集。而复常数是无穷的，因此说茹利亚集也是无穷的。为叙述的方便，我们用符号 J_c 表示复数 c 对应的茹利亚集。

我们说茹利亚集是分形，是因为它的边界线上的几何维数不是整数。图 3.6 是某个 J_c 在点 (0, 1) 的邻域，其半径逐级缩减再经放大后所显示的曲线图形，看得出它的几何维数在逐渐减少，但不会是整数。

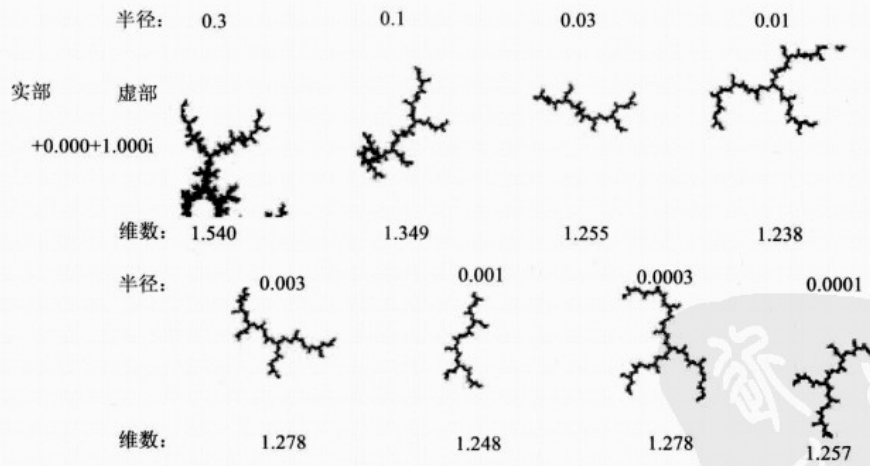


图 3.6　茹利亚集上某点不同半径范围内的图形维数

M-集和 J_c 都源于映射 $z_{n+1} = z_n^2 + c$，它们有天然的“血缘”关系。图 3.7 显示了 M-集的某一点所对应的 J_c。看得出，愈是相邻的点，其图形就愈是相近。

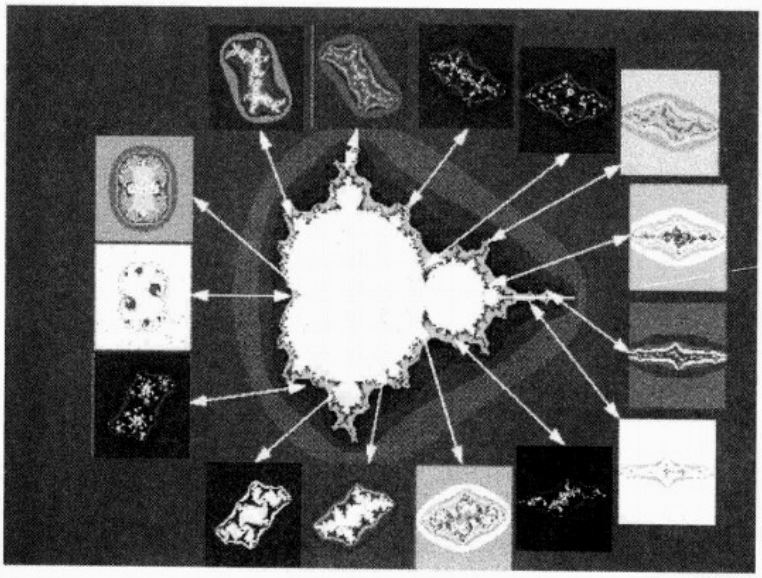


图 3.7　M-集上的点 c 所对应的 J_c

图 3.8 则显示了某个 J_c 上的若干点 c 所对应的不同 J_c。

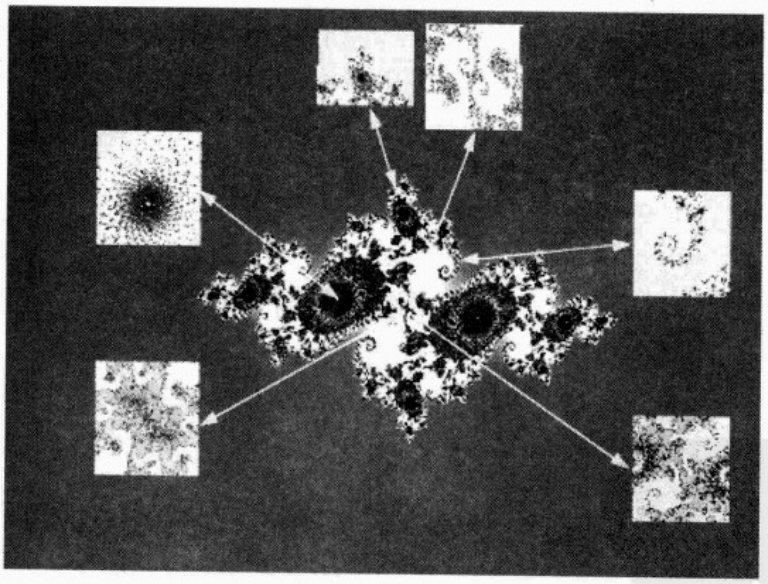


图 3.8　J_c 上的点 c 所对应的 J_c

若记荧屏上点（像素）的一个茹利亚迭代为 f，则图 3.9 显示了这样一个事实：$f(A)=B$，$f(B)=C$，$f(C)=D$，$f(D)=E$，$f(E)=A$。按定义 2.2 的说法，A 是周期 4 点，图上的箭线表示从 A 出发的一个轨道，也就是一个循环。

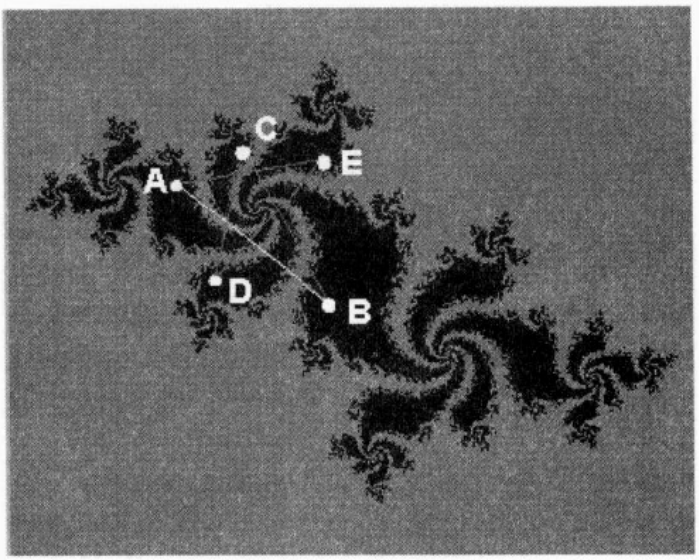


图 3.9　J_c 上的一个循环

第 2 章所提的米修雷维茨点是很有意思的点，谭蕾证明了这样一个事实：围绕 M-集的每个米修雷维茨点 c 的周围区域，放大以后看起来很像 J_c 围绕 c 点的一个适当放大区域。当这种放大继续提升以后，M-集与 J_c 的区域看起来就越来越像，所以许多 J_c 的细节图就是包围 M-集边界的图。图 3.10 中，左上的 c 点就是 M-集的一个米修雷维茨点，右上的 J_c 就是该 c 点所对应的茹利亚集。下方两图从左到右分别是 c 对应区域放大的 M-集与 J_c 集。它们是何等地相像！

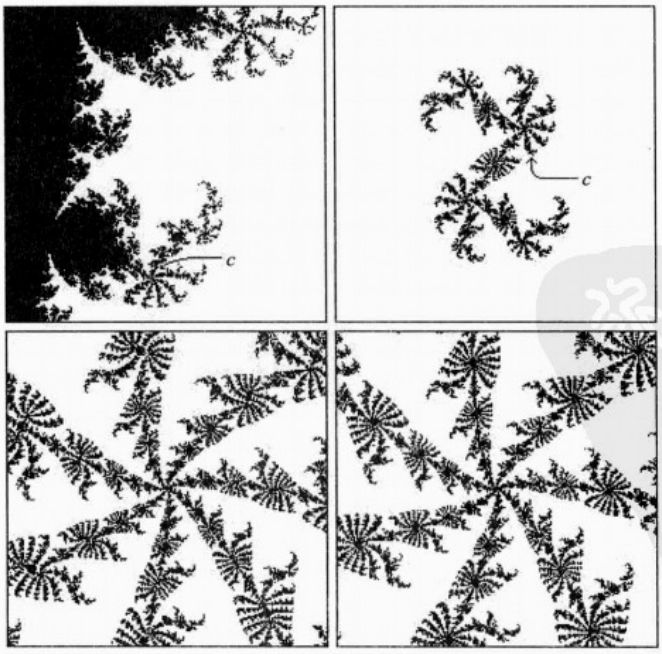


图 3.10　M-集的米修雷维茨点与其 J_c 集

在第 2 章，我们曾经把 M-集的某一点域一再放大并且数次见到小的 M-集。但由于茹利亚集千变万化，我们很难从它的局部累次放大去找出它的复制品来。图 3.11 是某茹利亚集局部一再放大的显示图，虽然我们难以找出其原图的克隆，但它们彼此之间的花纹的确很像。

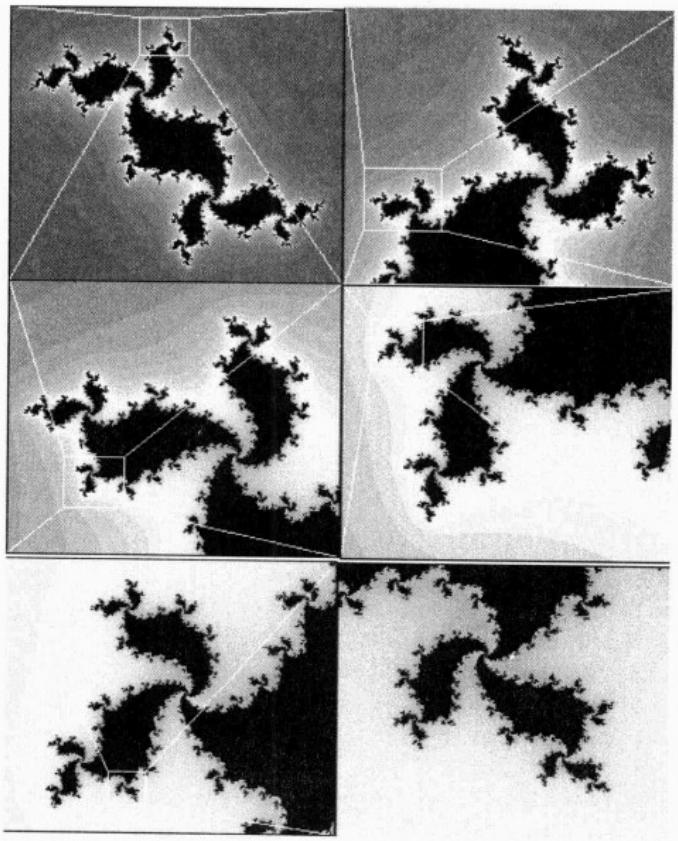


图 3.11　茹利亚集的局部累次放大图

程序 3.1 是绘画茹利亚集的 Java Applet 程序，可以很方便地调整参数画出各类 J_c 集。

程序 3.1　J_c 集的 Java 程序。

```
package julia;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class Complex {
  double x, y;
  public Complex() {
  }
  public Complex(double _x, double _y){
 x=_x; y=_y;
  }
  public Complex multiply(Complex z){
```

```
    return new Complex(x*z.x - y*z.y, y*z.x + x*z.y);
  }
  public Complex substract(Complex z){
    return new Complex(x-z.x, y-z.y);
  }
  public Complex add(Complex z){
    return new Complex(x+z.x, y+z.y);
  }
  public double norm(){
    return Math.sqrt(x*x + y*y);
  }
}

public class main extends Applet {
  boolean bcolor=false;
  final int OFFSET=30;
  final int SCREENWIDTH=400;
  final int SCREENHEIGHT=300-OFFSET;
  double xmax=2, xmin=-2, ymax=2, ymin=-2, size=4;
  int max_iterations=20;
  boolean bDraw=false;
  Complex c;

  private Label label1 = new Label();
  private TextField textField1 = new TextField();
  private Label label2 = new Label();
  private TextField textField2 = new TextField();
  private Button btnDraw = new Button();
  private Label label3 = new Label();
  private Label label4 = new Label();
  private Checkbox chkColor = new Checkbox();

  //Get a parameter value

  //Construct the applet
  public main() {
  }
  //Initialize the applet
  public void init() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  //Component initialization
  private void jbInit() throws Exception {
    label1.setText("c:");
    textField1.setText("1.0");
```

```
    label2.setText("+");
    textField2.setText("0.0");
    btnDraw.setLabel("画图");
    btnDraw.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        btnDraw_actionPerformed(e);
      }
    });
    this.setBackground(Color.white);
    label3.setText("i");
    label4.setText("f(z)=z*z+c");
    chkColor.setBackground(Color.white);
    chkColor.setLabel("彩色");
    chkColor.addItemListener(new java.awt.event.ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        chkColor_itemStateChanged(e);
      }
    });
    this.add(label4, null);
    this.add(label1, null);
    this.add(textField1, null);
    this.add(label2, null);
    this.add(textField2, null);
    this.add(label3, null);
    this.add(chkColor, null);
    this.add(btnDraw, null);

    c=new Complex(1, 0);
  }
  //Start the applet
  public void start() {
  }
  //Stop the applet
  public void stop() {
  }
  //Destroy the applet
  public void destroy() {
  }
  //Get Applet information
  public String getAppletInfo() {
    return "Applet Information";
  }
  //Get parameter info
  public String[][] getParameterInfo() {
    return null;
  }

  public void paint(Graphics g){
    g.setColor(Color.blue);
```

```
  if (bDraw) drawJulia(g);
  bDraw=false;
}

void btnDraw_actionPerformed(ActionEvent e) {
  /*if (rmin>=rmax || rmin<0 || rmax>4) {

    return;
  }*/
  bDraw=true;
  repaint();
}

double strtofloat(String str) {
  int sgn=1;
  int i=0, j, k=0;
  int len1=0, len2=0;
  char str1[], str2[];
  str1= new char[10];
  str2= new char[80];

  while (str.charAt(i)==' ')
    i++;
  if (str.charAt(i)=='-') {
    sgn=-1; i++;
  }
  if (str.charAt(i)=='+')
    i++;
  while (i<str.length() && str.charAt(i)!='.') {
    str1[len1++]=str.charAt(i);
    i++;
  }
  i++;
  while (i<str.length() && str.charAt(i)!=' ') {
    str2[len2++]=str.charAt(i);
    i++;
  }
  double sum=0;
  double e=Math.pow(10,len1-1);
  for (j=0; j<len1; j++) {
    sum+=(str1[j]-'0')*e;
    e/=10;
  }
  e=0.1;
  for (j=0; j<len2; j++) {
    sum+=(str2[j]-'0')*e;
    e/=10;
  }
  return sgn*sum;
}
```

```
Color GetColor(int c) {
   switch (c){
    case 0:
      return Color.black;
    case 1:
      return Color.blue;
    case 2:
      return Color.green;
    case 3:
      return Color.cyan;
    case 4:
      return Color.red;
    case 5:
      return Color.magenta;
    case 6:
      return Color.yellow.darker().darker();
    case 7:
      return Color.lightGray;
    case 8:
      return Color.darkGray;
    case 9:
      return Color.blue.brighter();
    case 10:
      return Color.green.brighter();
    case 11:
      return Color.cyan.brighter();
    case 12:
      return Color.red.brighter();
    case 13:
      return Color.magenta.brighter();
    case 14:
      return Color.yellow;
    case 15:
      return Color.white;
    default:
      return Color.white;
  }
}

void drawJulia(Graphics g){
  double x, y, dx, dy;
  int row, col;
  int color;

  setConst();
  dx = (xmax-xmin)/SCREENWIDTH; dy = (ymax-ymin)/SCREENHEIGHT;
  for (col=0; col<SCREENWIDTH; col++)
  {
    x=xmin+col*dx;
```

```
        for (row=0; row<SCREENHEIGHT; row++)
        {
          y=ymax-row*dy;
          color=0;
          Complex z=new Complex(x, y); //Init z
          while (color < max_iterations && z.norm() < size)
          {
            //z=f_sierpinski(z);
            z = z.multiply(z).substract(c);
            color++;
          }
          if (bcolor){
            if(color>=max_iterations)
              color=0;
            else
              //color=0;
              color=color%15+1;
          }
          else
          {
            if(color>=max_iterations)
              color=0;
            else
              color=15;
          }
          g.setColor(GetColor(color));
          g.drawOval(col, OFFSET+row, 1,1);
        }
      }
    }
    void setConst(){
      xmax=2; xmin=-2; ymax=2; ymin=-2; size=4;
      max_iterations=20;
      c=new Complex();
      c.x=strtofloat(textField1.getText());
      c.y=strtofloat(textField2.getText());
    }

    void chkColor_itemStateChanged(ItemEvent e) {
      bcolor=!bcolor;
    }
  }
```

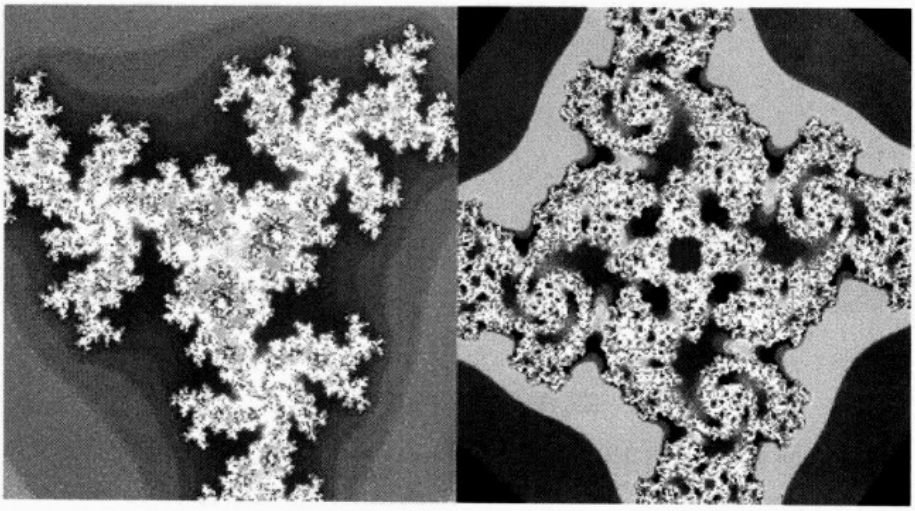
3.2　广义茹利亚集

3.2.1　整数次幂的广义茹利亚集

把公式（3.1）转换成

$$z_{n+1}=z_n^p+c,\ z_n,\ c\in C,\ p\in \mathbb{Z} \tag{3.3}$$

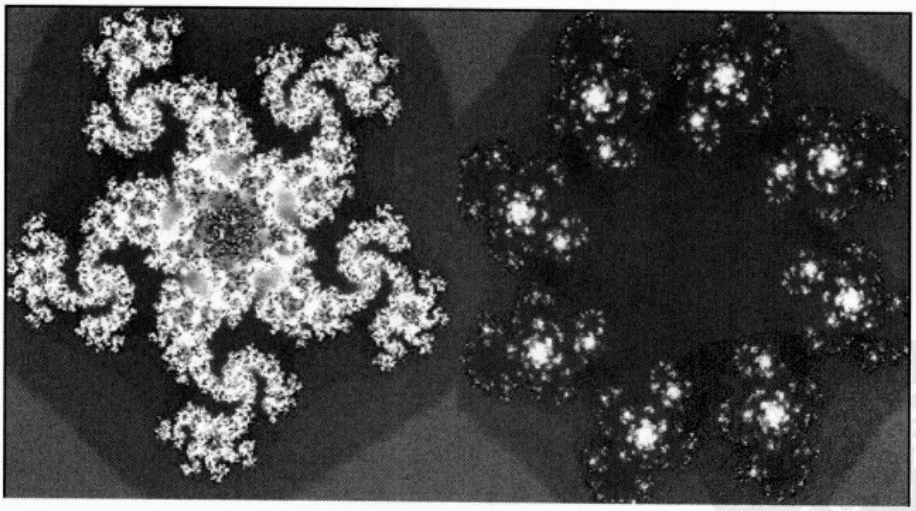
当 $p=3$ 且令 $c=(0.3330, -0.7122)$，得到图 3.12(a)，$p=4$ 且 $c=(-0.6630, -0.3522)$ 时，得到图 3.12(b)。



(a)　　(b)

图 3.12　三次和四次广义 J_c 集

令 $p=5$ 且 $c=(0.32, 0.7)$，得到图 3.13(a)，$p=8$ 且 $c=(0.6711, 0.5000)$，得到图 3.13(b)。



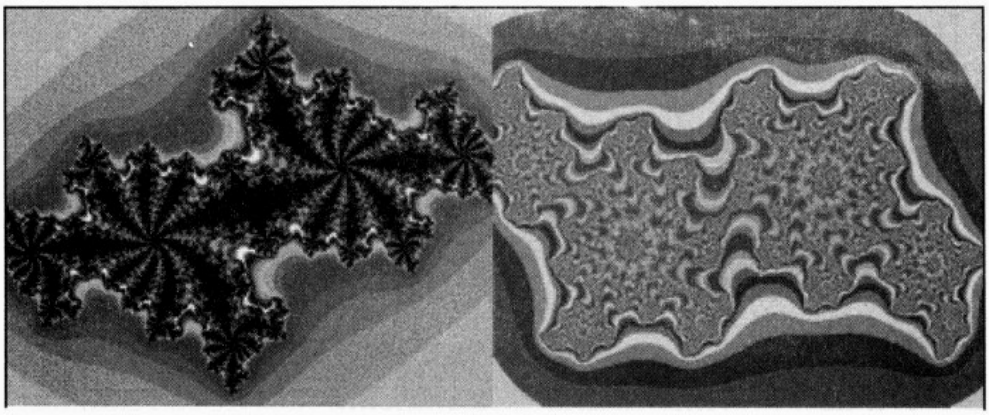
(a)　　(b)

图 3.13　五次和八次广义 J_c 集

也能在整数次幂函数后面加修正项的迭代，一种公式是

$$z_{n+1}=z_n^2+f(c)+p \quad z_n, c, p\in C \tag{3.4}$$

当 $f(c)=\tan(c)$ 且 $c=(-0.17, -1.9)$，$p=(-0.67, 0.62)$时，画得图 3.14(a)。当 $f(c)=\arcsin(c)$，$c=(1.33, -0.70)$，$p=(-0.90, 0.40)$时，得到图 3.14(b)。



（a）　　（b）

图 3.14　整数次幂加修正项的广义茹利亚集之一

当 $f(c)=\sin(c)$，$c=(1.19,\ -0.83)$，$p=(-0.89,\ 0.45)$时，见图 3.15。

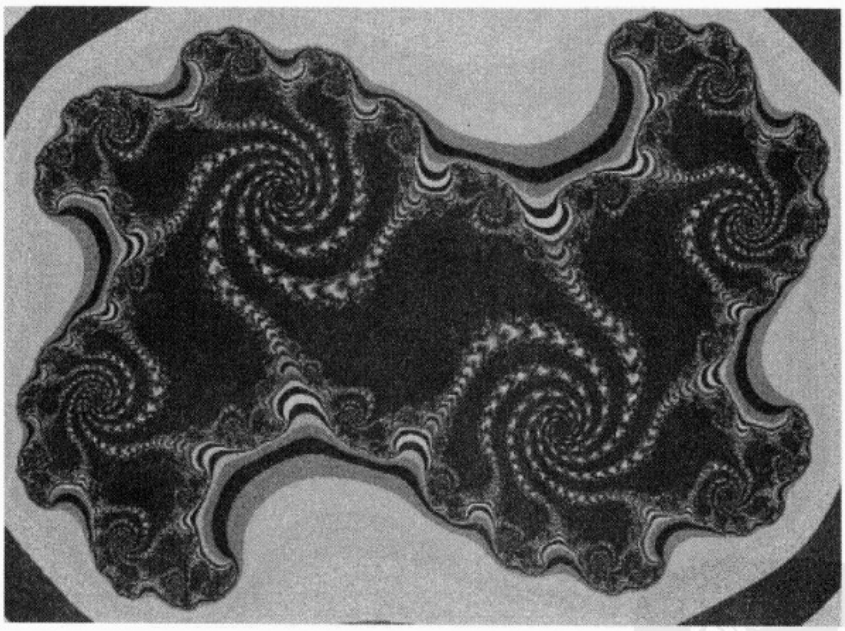


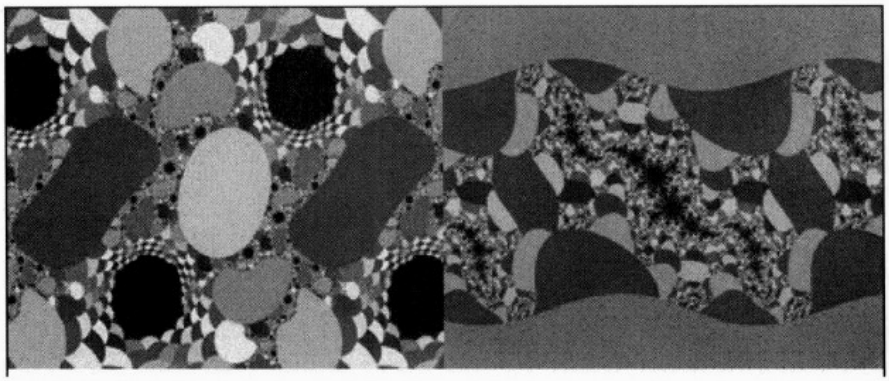
图 3.15　整数次幂加修正项的广义茹利亚集之二

3.2.2　其他函数代替 z^2 的广义茹利亚集

迭代格式 1 是

$$z_{n+1}=f_1^2(z_n)+f_2(c)+p \qquad z_n, c, p\in C \tag{3.5}$$

分别选定 $f_1(z)=\mathrm{c}\tan(z)$，$f_2(c)=c$，且 $c=(-0.15,\ -1.04)$，$p=(0.08,\ 0.53)$，和 $f_1(z)=\cos z$，$f_2(c)=\mathrm{c}\tan(c)$，且 $c=(-0.14,\ -1.03)$，$p=(0.08,\ -0.04)$，则分别得到图 3.16(a)和 3.16(b)。



（a）　　（b）

图 3.16　其他函数代替 z^2 的广义茹利亚集之一

迭代格式 2 是

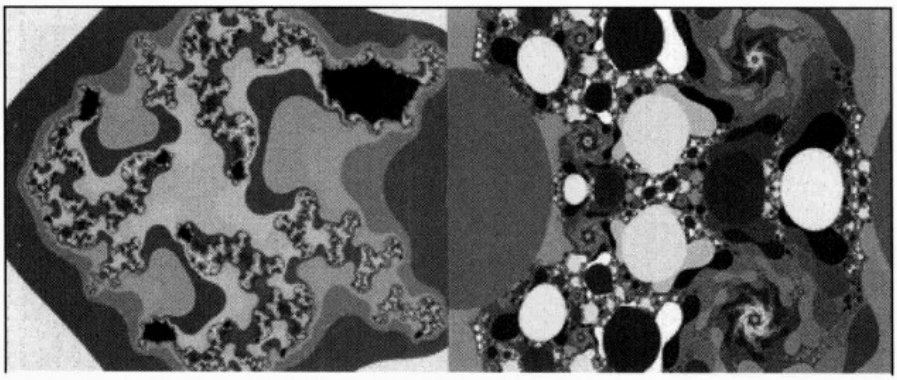
$$z_{n+1} = z_n^3 + f_1(z_n^2) + f_2(c),\ z_n,\ c \in C \tag{3.6}$$

令 $f_1(z) = \cos(z)$，$f_2(c) = \mathrm{c}\tan(c)$，$c = (-0.24, -0.21)$，得到图 3.17(a)，

迭代格式 3 是

$$z_{n+1} = f_1^2(z_n) + f_2(c),\ z_n,\ c \in C \tag{3.7}$$

令 $f_1(z) = \mathrm{c}\tan(z)$，$f_2(c) = e^c$，$c = (-0.78, -0.01)$，见图 3.17（b）。



（a）　　（b）

图 3.17　其他函数代替 z^2 的广义茹利亚集之二

迭代格式 4 为

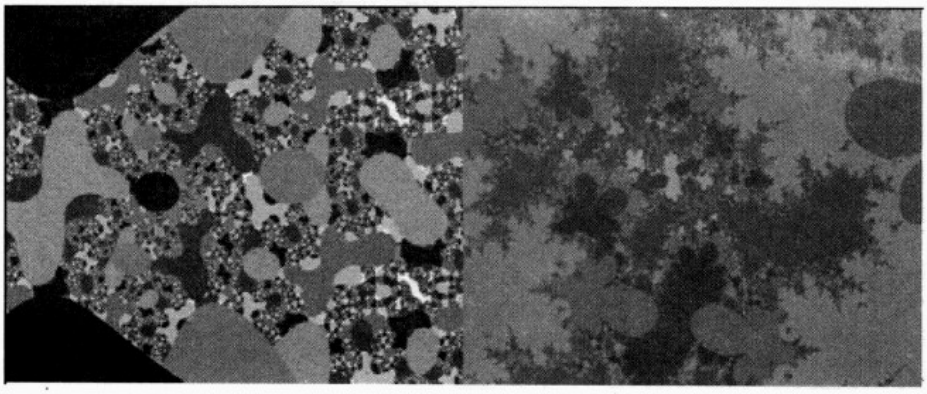
$$z_{n+1} = \frac{f_1^2(z_n)}{f_1(z_n) + p} + c,\ z_n,\ p,\ c \in C \tag{3.8}$$

令 $f_1(z) = \sin(z)$，$c = (-0.92, -0.09)$，$p = (0.48, -0.03)$，产生的图见图 3.18(a)。

迭代格式 5 是

$$z_{n+1} = f_1^2(z_n) + c \cdot \sin(z_n + p_1) f_2(z_n^2(z_n + c)) + c \cdot p_2,\quad z_n, p_1, p_2, c \in C \tag{3.9}$$

令 $f_1(z) = \cos(z)$，$f_2(z) = \sin(z)$，$c = (0.5602, 0.3895)$，$p_1 = (-0.95, 0.73)$, $p_2 = (0.56, 0.48)$，则得到图 3.18(b)。



(a)　　(b)

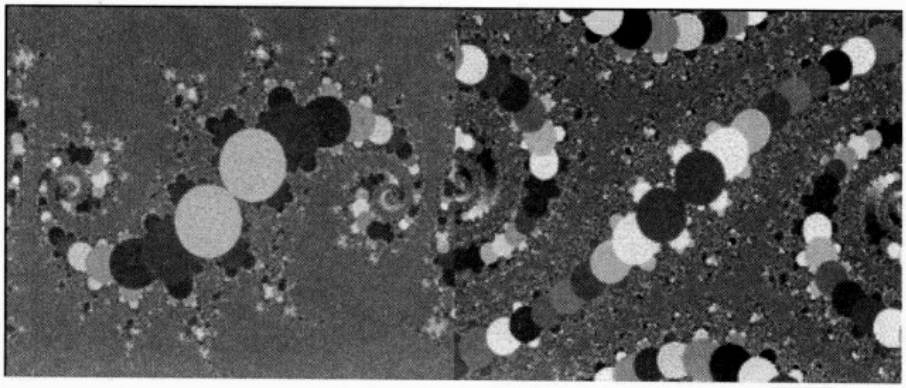
图 3.18　对其他函数代替 z^2 的广义茹利亚集之三

迭代格式 6 为

$$z_{n+1}=\sin(\frac{f_1(c)}{\overline{z}_n}),\qquad z_n,c\in C \tag{3.10}$$

式中符号 $\overline{z}_n$ 表示 z_n 的共轭。

分别令 $f_1(c)=\cos(c)$，$c=(0.45,\ -0.41)$和 $f_1(c)=\sin(c^2)$，$c=(0.43,\ -0.424)$，则分别得到图 3.19(a)和图 3.19(b)。



(a)　　(b)

图 3.19　其他函数代替 z^2 的广义茹利亚集之四

总之，上述采用的方法由于初等函数的无限性，其产生的图像当然也是无穷无尽的。

3.2.3　牛顿分形

一个连续可微函数 $f(x)$，如果取其 Taylor 展开前两项作为它的近似，则有

$$f(x)\approx f(x_0)+f'(x_0)(x-x_0)$$

令 $f(x)=0$，则函数 $f(x)$ 的零点 x 可表示为

$$x\approx x_0-\frac{f(x_0)}{f'(x_0)}$$

于是有了求 $f(x)=0$ 方程根的牛顿（Newton）迭代法

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)} \qquad n=0,1,2,\cdots \tag{3.11}$$

如果 $\lim\limits_{n\to\infty}x_n$ 存在，则序列 $\{x_n\}$ 平方收敛到 $f(x)$ 的零点。

我们把复数 z 运用到式（3.11）上，得到如下公式：

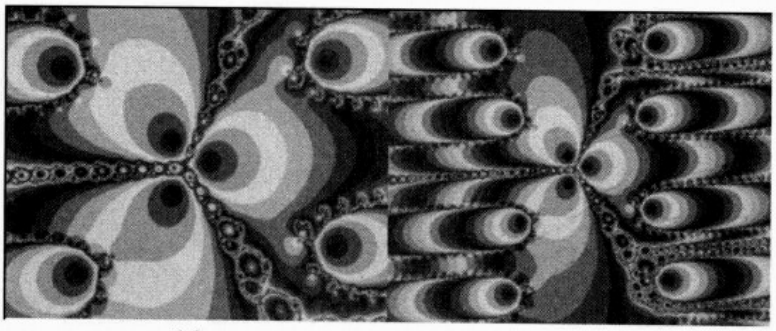
$$z_{n+1}=z_n-\frac{f(z_n)}{f'(z_n)} \qquad n=0,1,2,3,\cdots \tag{3.12}$$

按茹利亚绘图法则并运用逃逸时间法就画得一幅牛顿分形图。

设 $f(z)=z^3+cz^2-z-z\sin z+c$，则有迭代公式

$$z_{n+1}=z_n-\frac{f(z_n)}{3z_n^2-2cz_n-\sin z_n-z_n\cos z_n-1},\quad z_n,c\in C \tag{3.13}$$

令 $c=(-0.36,2.336)$，得到图 3.20（a），图 3.20（b）是该图放大四倍后中心区域的图。



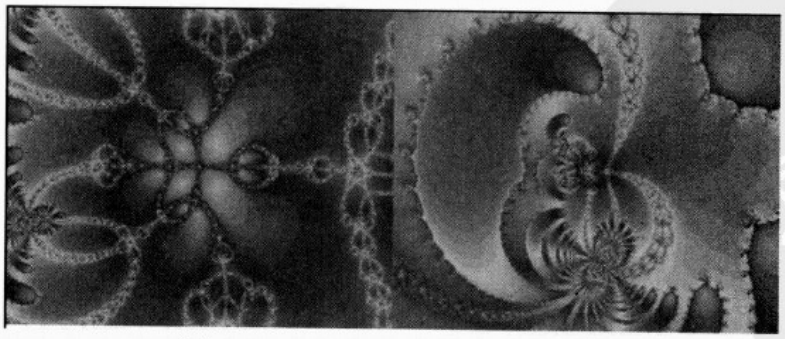
（a）　　（b）

图 3.20　三次函数牛顿分形之一

设 $f(z)=cz^4+z^3-z\sin z+z+c$，则其牛顿迭代公式为

$$z_{n+1}=\frac{f(z_n)}{4cz_n^3+3z_n^2-\sin z_n-z_n\cos z_n+1},\quad z_n,c\in C \tag{3.14}$$

令 $c=(-4.560, 0.036)$，得到图 3.21(a), 3.21(b)是其某一局部区域放大八倍后的图形。



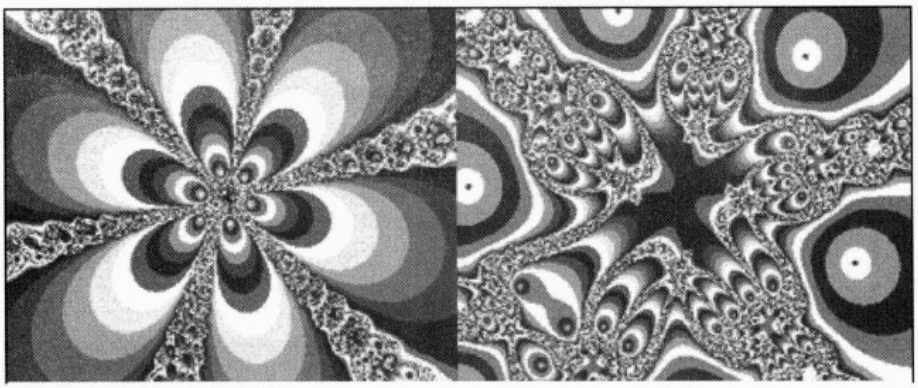
（a）　　（b）

图 3.21　四次函数牛顿分形

设 $f(z)=z^6+cz^5+\sin z+p$，其牛顿公式应是

$$z_{n+1}=\frac{f(z_n)}{6z_n^5+5cz_n^4+\cos z_n},\qquad z_n, p, c\in C \tag{3.15}$$

令 $c=(1.6,\ 1.16)$，$p=(0.92,\ 15.53)$，其对应的图见图 3.22(a)，图 3.22(b)是中心区域放大八倍后得到的图形。



(a)　　(b)

图 3.22　六次函数牛顿分形

下面是另一种三次和八次牛顿分形，见图 3.23 和图 3.24。

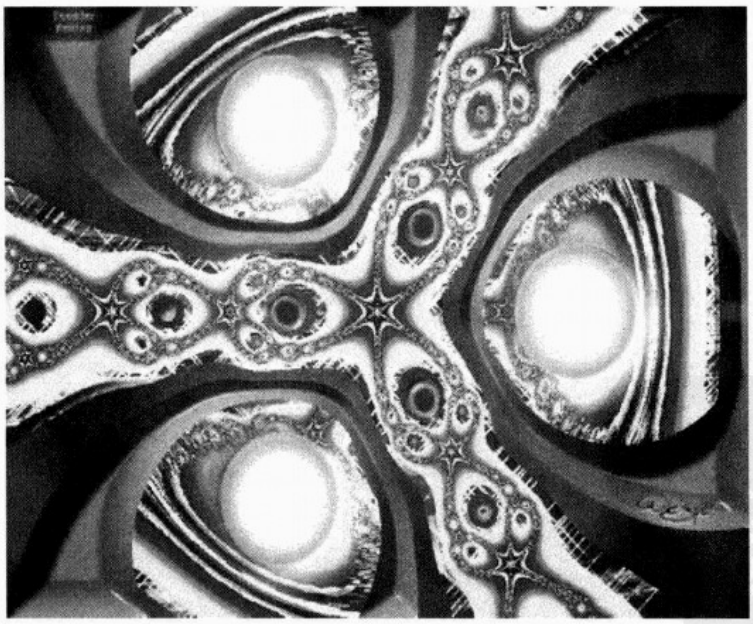


图 3.23　三次函数牛顿分形之二

程序 3.2 是一个三次多项式牛顿法的 Maple 程序及其做出的画，见图 3.25。

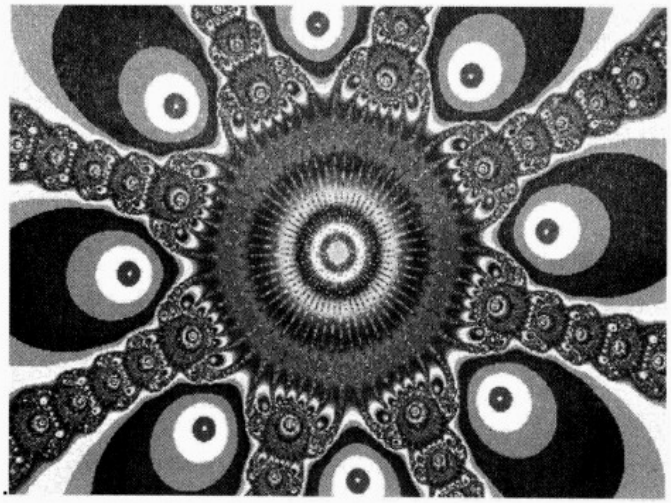


图 3.24　八次函数牛顿分形

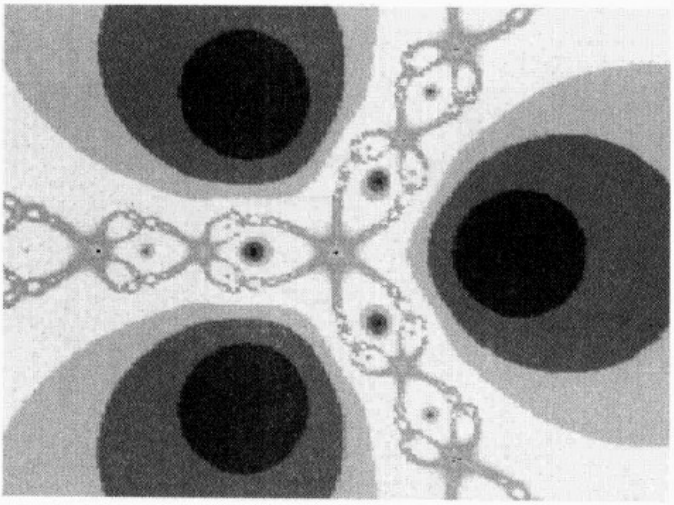


图 3.25　三次多项式牛顿分形

程序 3.2　牛顿分形的 Maple 程序。

```
> restart:

> newton := proc(x, y)
>    local z, m;
>    z := evalf(x+y*I);
>    for m from 0 to 50 while abs(z^3-1) >= 0.001 do
>       z := z - (z^3-1)/(3*z^2)
>    od;
>    m
> end:

> plot3d(0, -2 .. 2, -1.5 .. 1.5, orientation=[-90,0],
>    grid=[250, 250], style=patchnogrid, scaling=constrained,
>    color=newton);
```

3.2.4　米拉分形

米拉（Mira）公式又称顾莫乌斯基-米拉（Gumowski-Mira）公式

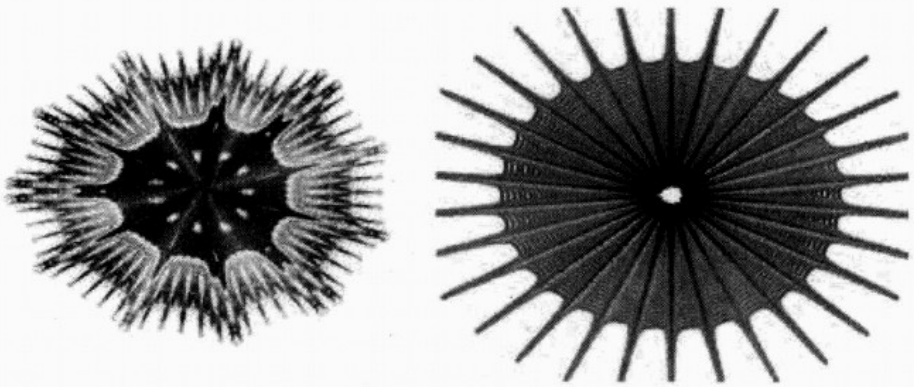
米拉公式作为下面的迭代法出现

$$\left.\begin{aligned} x_{n+1} &= by_n + f(x_n) \\ y_{n+1} &= f(x_{n+1}) - x_n \end{aligned}\right\} \tag{3.16}$$

式中 b 用于控制轨道是膨胀还是收缩。如果 b 稍大于 1，比如 1.004，则轨道就会膨胀，反之若稍小于 1，比如 0.998，轨道就将收缩。在对下面这些函数做图时，大多数用的是 1.005。一个经典的函数是

$$f(x) = ax + \frac{2(1-a)x^2}{1+x^2}$$

分别令 x =14.717684295，y =–13.660209039，a =–0.662358470，b =1.00000，和 x=19.945948645，y=4.749808544，a=-0.669105405，b=1.00001，做图得到图 3.26(a)和图 3.26(b)。



(a)　　(b)

图 3.26　米拉分形之一

下面这些函数是常用的：

$$f(x) = ax + 2sign(x)(1-a)x^2/(1+x^2)$$
$$f(x) = sign(x)ax + (1-a)x^2/(1+x^2) + \sin(x)$$
$$f(x) = sign(x)(ax + 2(1-a)x^2/(1+x^2)) + \sin(a+x)$$
$$f(x) = -0.05ax + a(\pi - ax)x^2/(1+x^2)$$
$$f(x) = asign(x) + ax\sin(x+\pi)$$
$$f(x) = asign(x) + \sin(x)\sin(x+\pi)$$
$$f(x) = asign(x) + \sin(1/x)$$
$$f(x) = 2\pi acsign(x) + \pi + \sin(\pi + \sin(\pi + x))$$
$$f(x) = 2\pi acsign(x) + \pi + \cos(x) + \sin(\pi + \sin(\pi + x))$$

式中函数 $sign(x)$=如果 $x \geqslant 0$，则 1，否则–1。若干样图见图 3.27 和图 3.28。

米拉公式算法：（w 代表函数，random 是随机数且 0<random<1）

初值

```
a=2(random-0.5)
b=1.005
w=1
x=0.0004
y=0.0004
```

计算轨道（在迭代 100 到 20000 次间，每次都画 (x, y) 点，当然也可加进颜色）

```
z=x
x=by+w
w=f(x)（如上所列的某一个）
y=w-z
plot(x,y)
```

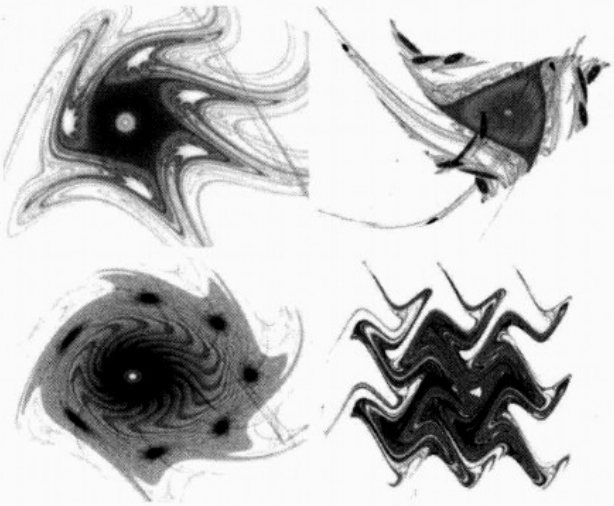


图 3.27　米拉分形之二

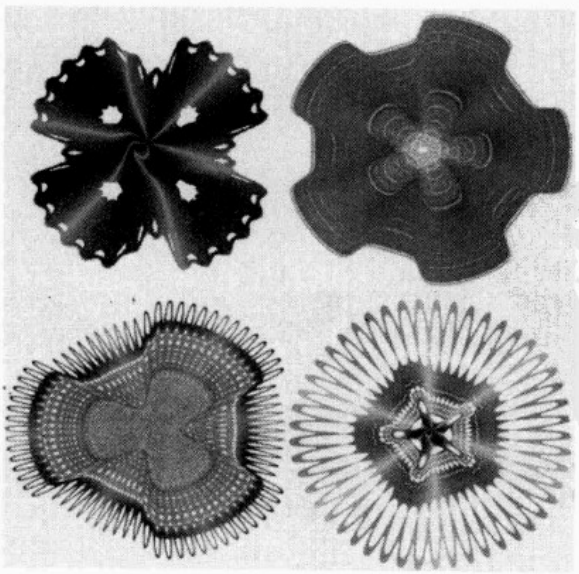


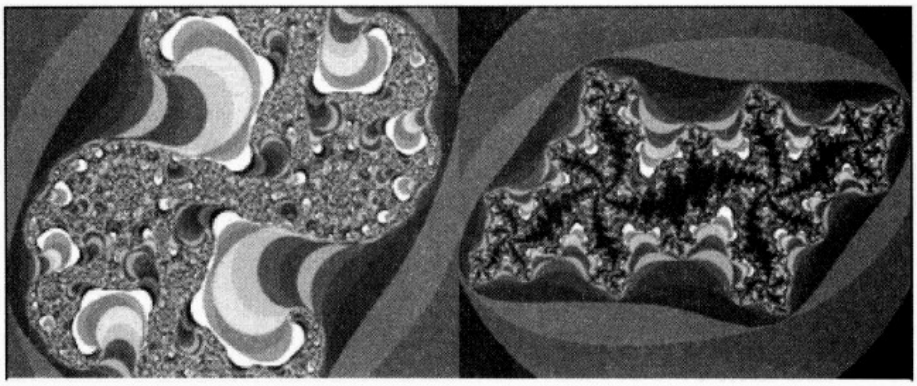
图 3.28　米拉分形之三

3.2.5　λ 分形

λ 分形的一种迭代格式为

$$z_{n+1} = \lambda z_n(1 - z_n), \quad \lambda, z_n \in C \tag{3.17}$$

式（3.17）又称逻基斯缔克（Logistic）公式，在混沌科学中占有重要位置。上式中，分别取 $\lambda = (0.8500, 0.5555)$ 和 $\lambda = (-0.85, 0.6)$ 便得到图 3.29（a）和图 3.29（b）。



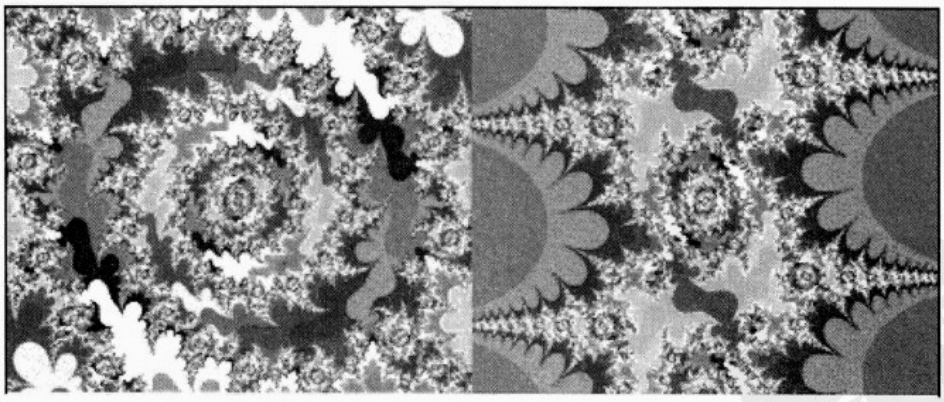
（a）　　（b）

图 3.29　λ 分形之一

λ 分形的另一种迭代格式为

$$z_{n+1} = \lambda f(z_n), \quad \lambda, z_n \in C \tag{3.18}$$

分别取 $\lambda = (1, 0.4)$，$f(z) = \sin z$ 和 $\lambda = (1, -0.4)$，$f(z) = \sinh z$，则得到图 3.30(a)与图 3.30(b)。



（a）　　（b）

图 3.30　λ 分形之二

λ 分形的第三种格式是：如果 $|z_n| < M$，则

$$z_{n+1} = \lambda f_1(z_n)$$

否则

$$z_{n+1} = \lambda f_2(z_n)$$

取 $M=1, \lambda=(1, 0.1)$，$f_1(z)=\sin z$，$f_2(z)=\sqrt{z}$，得到图 3.31。

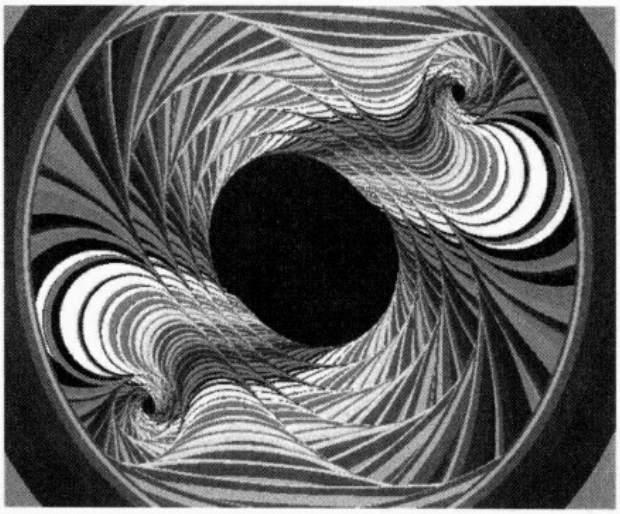
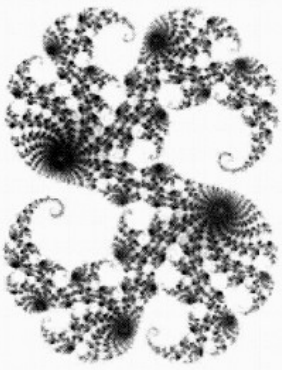


图 3.31　λ 分形之三

3.3　伪三维茹利亚分形

和曼德勃罗集一样，本章介绍的并不是真正的三维立体茹利亚集，真正的三维立体茹利亚集将在第 6 章的四元数分形中叙述。这里介绍的仅是 2D 茹利亚集的立体化，故称伪三维茹利亚集。

第一种是把原图像人为加厚。下面图中，图左为二维原始图像，图右则是伪三维图，不过这里也有一些人为的加工，见图 3.32～图 3.35。图 3.35 则无原始图。



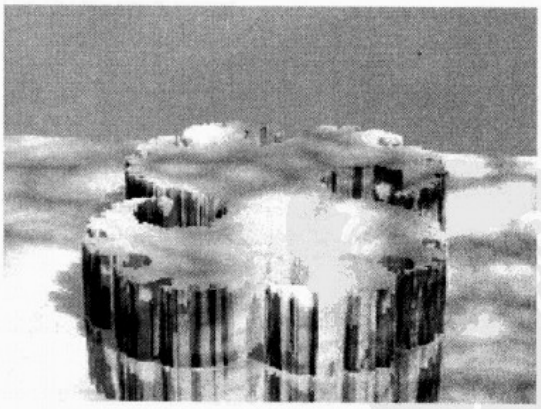


图 3.32　伪三维茹利亚集之一

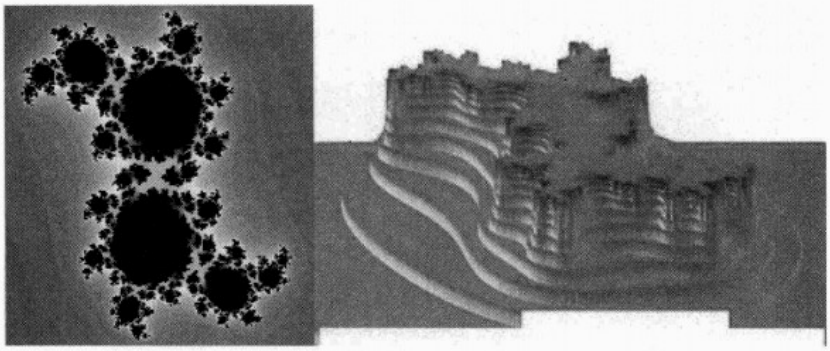


图 3.33　伪三维茹利亚集之二

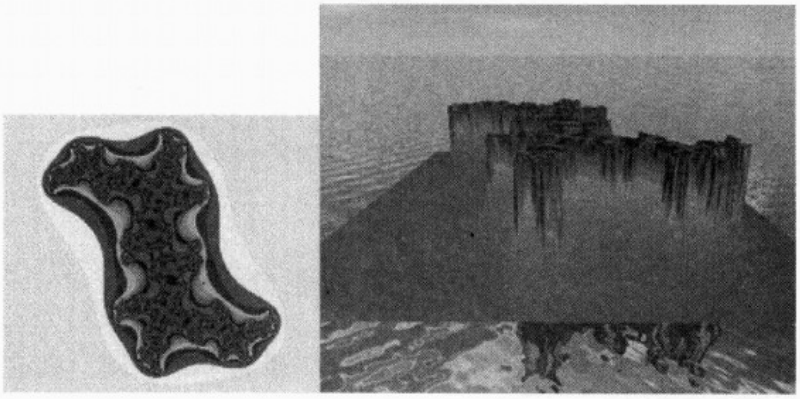


图 3.34　伪三维茹利亚集之三

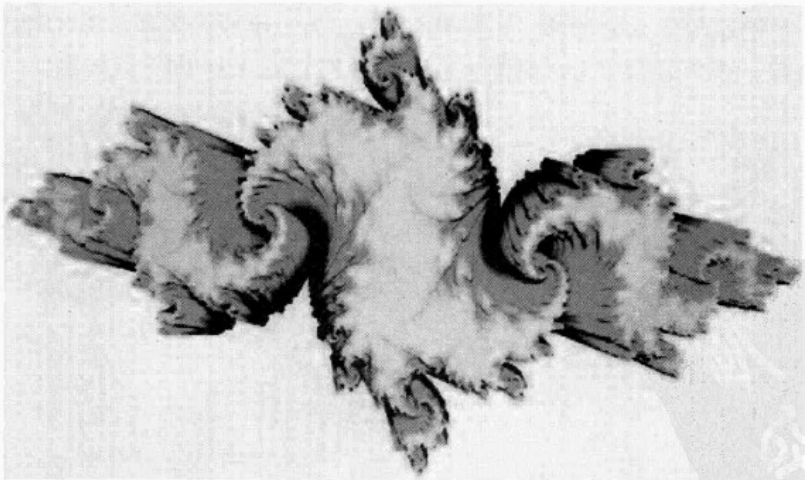


图 3.35　伪三维茹利亚集之四

还有一类是把原图雕刻成凹下去的样子，见图 3.36～图 3.37。

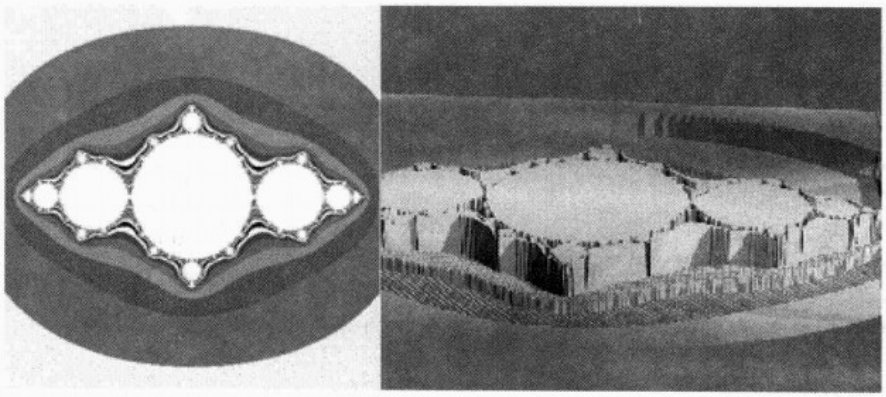


图 3.36　伪三维茹利亚集之五

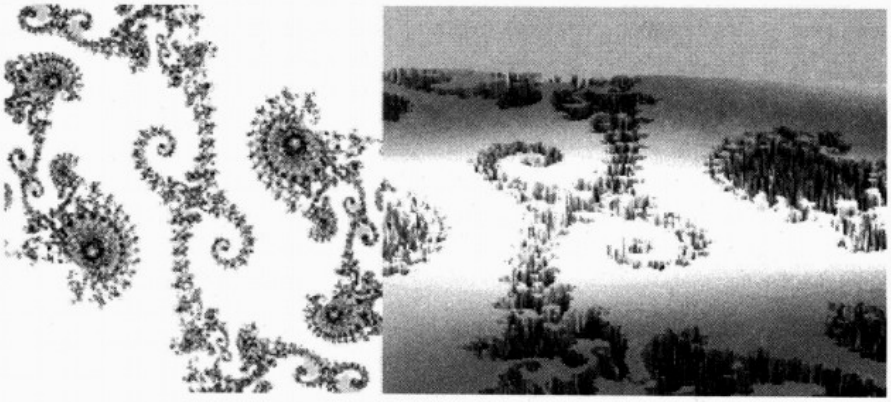


图 3.37　伪三维茹利亚集之六

图 3.38 是更艺术化的伪三维茹利亚集。



图 3.38　伪三维茹利亚集之七

更多的茹利亚集请见本书附带光盘。

第 4 章　L-系统

L-系统的第一个字母 L 源于美国生物学家 A.Lindenmayer（1929～1989）姓氏中的 L 字母。开始是作为描述植物的形态与生长的一种方法，继而发展成计算机图形学中一种模拟大自然景物的有效方法，当然是一种重要的分形生成方法。

4.1　二维 L-系统

L-系统是由若干符号组成的语言提示系统。符号用法各有所取，但对最常用的符号目前比较通用的是下面给出的这组符号解释：

F：按给定的长度向前画出一条线段；

f：按给定的长度向前移动（不画）一条线段；

+：左转（逆时针方向）一个给定的角度；

-：右转（顺时针方向）一个给定的角度；

|：反向 180°；

[：把当前画图格式放进栈里；

]：退出画栈,并回到左方括号“[”前的画图状态；

#：按线段宽度的增量要求增加线段的宽度；

!：按线段宽度的减量要求减少线段的宽度；

@：按线段宽度作半径画一个点；

>：按线段长度的比例因子乘线长；

<：按线段长度的比例因子除线长；

&：交换+和-的意义；

(：按转动角度的增量减少角度的转动量；

)：按转动角度的增量增加角度的转动量。

4.1.1　简单的向前生成元格式

一个人在城市的大街上行走，当走到一个十字路口时可能会左转前进或右拐前进，这种前进（F）左拐（+）或右拐（-）将伴随他走完全程到目的地，如果将无穷的嵌套结构引进到这种简单的行走模式，将会产生怎样的效果呢？

令前进符号为 F 且前进的步长是相等的，左拐即按角度增加方向（逆时针方向）记为“＋”号，右拐即按角度减少的方向（顺时针方向）记为“－”号，拐的角度为 δ，若令 δ=90°，则图 4.1 中前进的路线可记为

FFF-F-FF+F+F-F+F-F-FF-F+F-FFF

注意，图上的箭头表示方向，在正式图上不出现箭头符号。

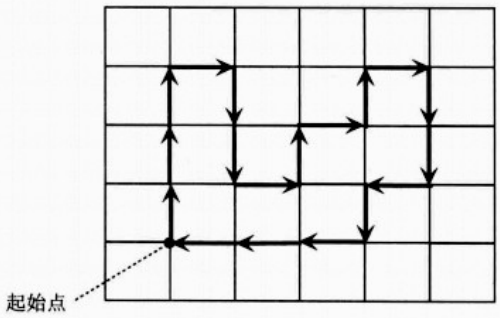


图 4.1　简单向前格式

为了得到一张分形图，需要下面几个要素：初始元、生成元、角度 δ 和压缩因子，并对程序规定，前进方向通常自左向右。

例 4.1　窗格子图。

初始元：F+F+F+F

角度 δ：90°

生成元：FF+F+F+F+FF

压缩因子 s：1/3

程序的第一步是产生初始元。按初始元规定，先从左边起始点按前进方向前进一单位长度，继而左转 90°，再前进一步，又左转 90°，再前进一步，再左转 90°后前进一步，最终得到边长为 1 的一个正方形，如图 4.2（a）所示。

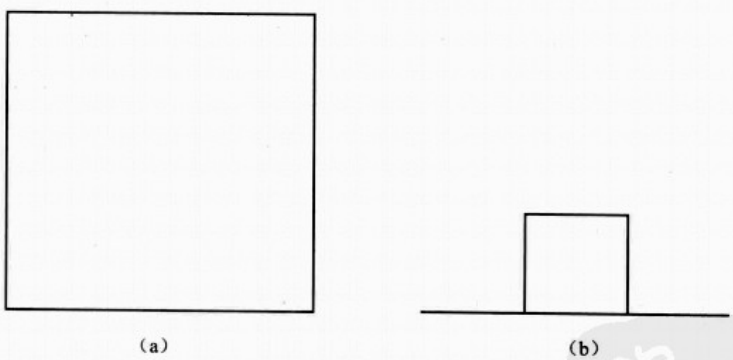


图 4.2　例 4.1 的初始元（a）和生成元的一步执行结果（b）

接下来的程序工作由生成元、角度 δ 和压缩因子联合进行：压缩原来步长到 1/3，从左边起始点开始对前进中的每条线段执行生成元格式。

FF：前进两步（原先长度的 2/3 长度）；

＋：原地左转 90°；

F：前进一步（原先长度的 1/3 长度）；

+：原地左转 90°；

F：前进一步；

＋：原地左转 90°；

F：前进一步；

+：原地左转 90°；

FF：前进两步。

结果是把原先正方形的一条边变成了图 4.2（b）所示图形，然后把正方形的其他三条边如法炮制，就得到图 4.3 的“level=2”图。迭代的第二步是把图 4.3 的“level=2”图的每条边再压缩步长 1/3，从原起始点开始对每条边执行生成元格式，就得到图中“level=3”所示图形，不断重复刚才过程，就得到一张 L-系统描绘的分形图。图 4.3 还显示了迭代 3 次和 4 次的图。

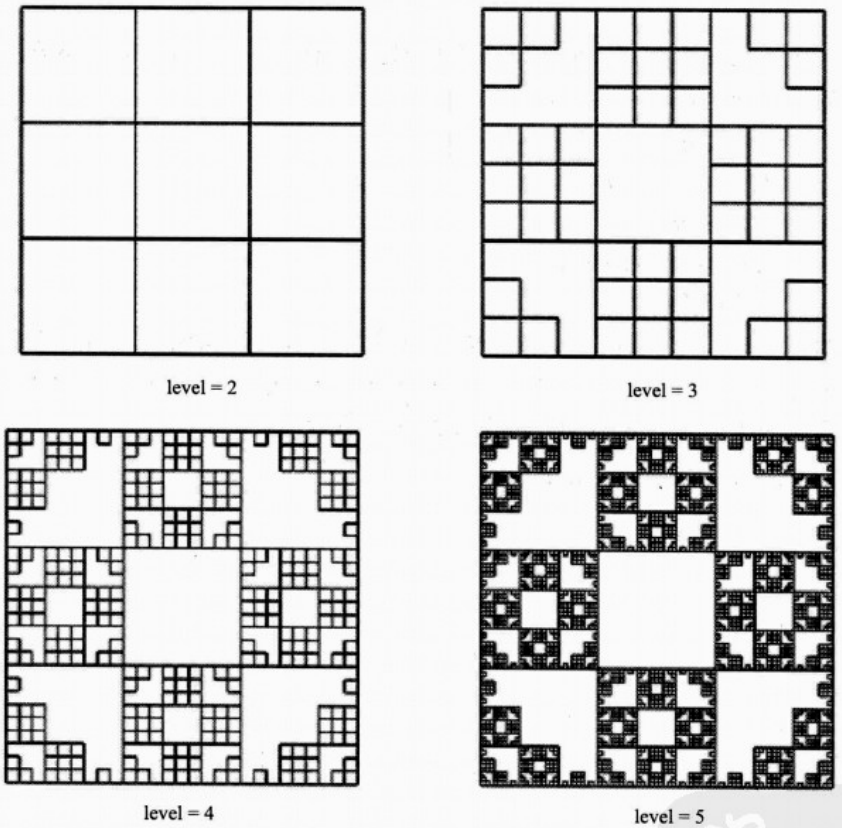


图 4.3　例 4.1 的执行结果

例 4.2　地蓆图。

初始元：F++F++F++F

生成元：F+++F---F+F---F++F—F++F

角度：45°

压缩因子：1/4

显然其初始元还是一个单位正方形。对正方形底边执行一次生成元，其图形见图 4.4。对四条边都执行生成元法则，其图形如图 4.5 的左上方所示。图 4.5 显示了前四次迭代结果。

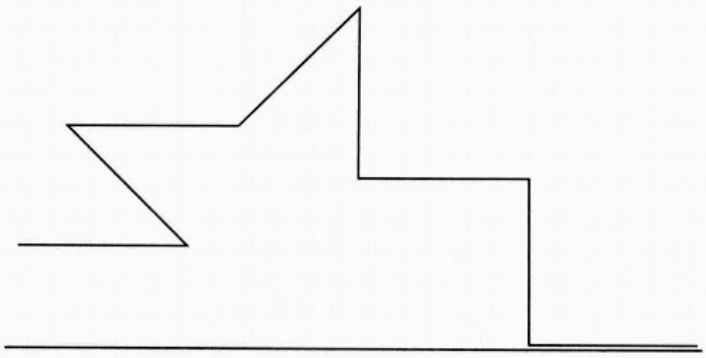


图 4.4　例 4.2 生成元的一次执行情况

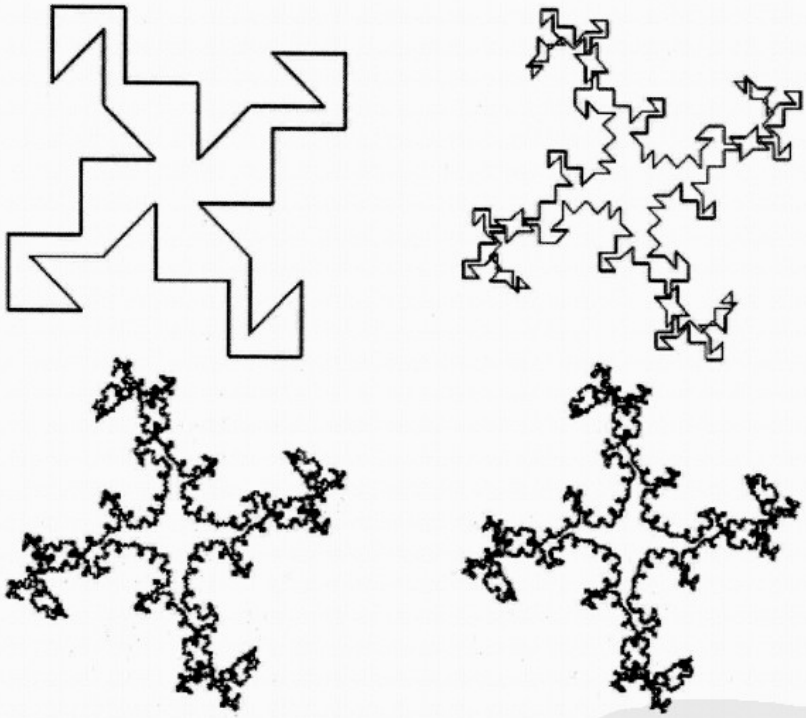


图 4.5　例 4.2 前 4 次执行结果

例 4.3　五角星。

初始元：F++F++F++F++F

生成元：F++F++F|F-F++F

角度：36°

压缩因子：0.38197

在生成元公式里有一个符号“|”，它表示反向 180°。

初始元是一个正五多边形，见图 4.7 的左上方。它的底边经生成元第一次迭代后，得到图 4.6。

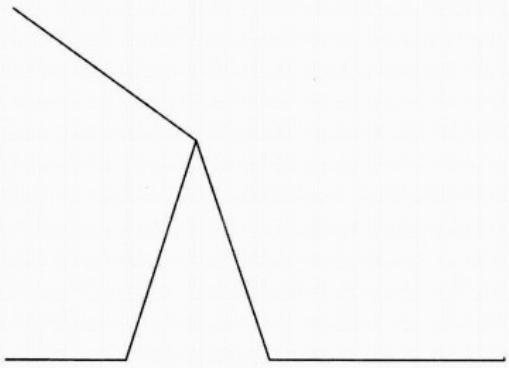


图 4.6　例 4.3 生成元的一次执行结果

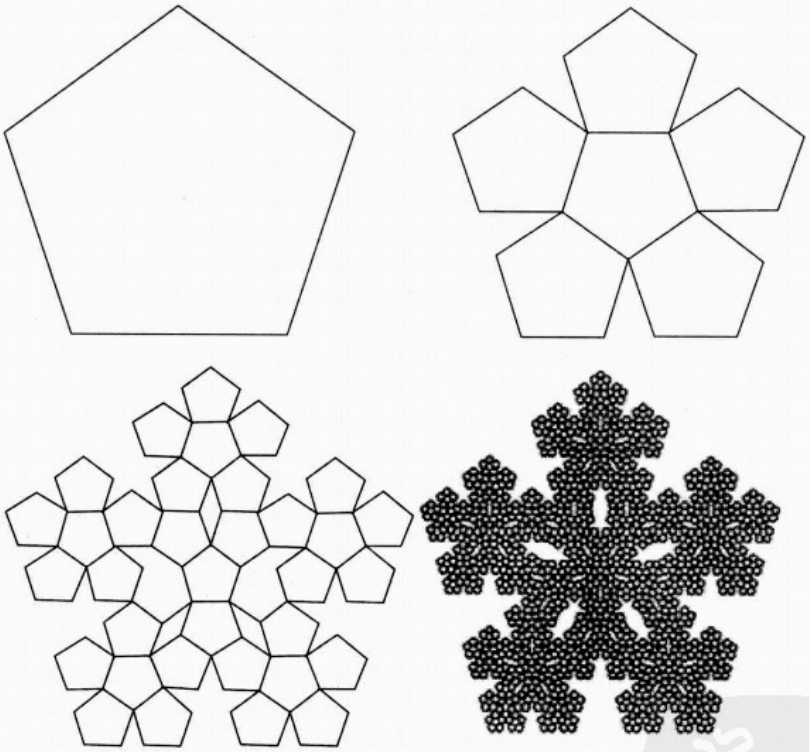


图 4.7　例 4.3 前 4 次执行结果

4.1.2　左右生成元的混合格式

人在街上走时，街边的景物与人行走的方向有个相对位置问题，当你从左向右走时，若国贸大厦在你的左边，则当你回头从右向左走时，该大厦却位于你右边位置了。

在上一节，我们只有一个向前生成元（或加进一个反向符号）。现引进左生成元和右生成元。记 L^*为左生成元，R^*为右生成元，每个生成元又是由左向走符号 L 和右向走符号 R 及“＋”和“－”组成。比如对于图像元素⊥，L 为⊥，则 R 便成了⊤。前者表示从左向右走，后者则是从右向左走的符号，因为计算机执行时已被规定从左向右

推进。例如

L^*：L+RR−L−L+L+RR+RRR−R−RRRR−RRRR

R^*：−LL+R−L+R−R+L+RLR+LL−R−RRR

δ=90°，压缩因子为 1/4。遂得图 4.8 所示图形。

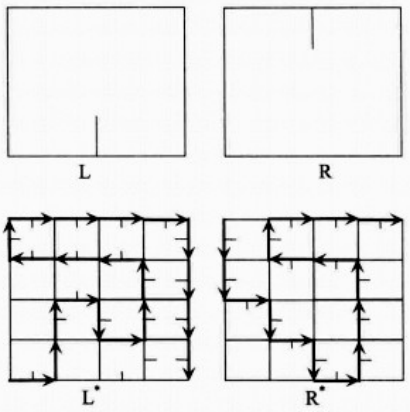


图 4.8　一种左右生成元图

要产生一张分形图，还需有一个初始元。一般来说，这儿的初始元比较简单，常以 R 或 L 表示，程序中 L^*与 R^*则分别用 L 和 R 代替。

例 4.4　龙图。

设初始元：L

左生成元 L：＋L－－R

右生成元 R：－L＋＋R

压缩因子：$1/\sqrt{2}$

角度 δ：45°

由于初始元为 L，所以一开始为一单位长线段，接下来的操作程序是先把这单位长线段按右生成元 R 运行，接着是左生成元 L，或者说，真正的初始元是 RL，见图 4.10（b）。但由于初始元为 L，所以以后每一次循环操作都按 LR 格式进行。

对给定的单位长线段，分别经 L：＋L－－R 和 R：－L＋＋R 运行后，得如图 4.9 中的两个图形（a）和（b）。

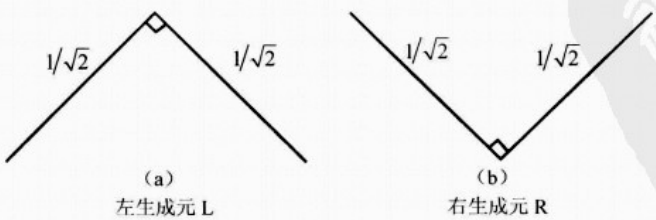


图 4.9　例 4.4 的左右生成元

在首次执行时，由于初始元是单位线段，所以在（＋L－－R）中，L 和 R 表面上看并无区别，它们都代表前进一步。

第一步，按右生成元 R 运行，得到图 4.10 中（a）。

第二步，对图 4.10（a）的第一条线段执行 R 生成元，即起始点右转 45°，线段压缩 $1/\sqrt{2}$ 倍后前进一步，再左转两个 45°，前进一步，得 L 状图形。接着对第二条边执行 L 生成元，得Γ状图形，于是有图 4.10（b）。

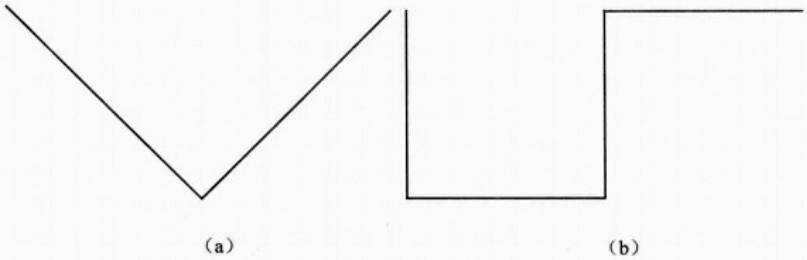


图 4.10　例 4.4 的第一、二步（第三级）图

第三步，对第三级（图 4.10（b））图形，从左到右，采取奇数段执行 L 生成元，偶数段执行 R 生成元，该例的第三步图见图 4.11 的左上图。

重复执行第三步，继多次迭代后，得到一个形似龙的图像，见图 4.11。图中显示了第 3、4、5 和第 15 步的图形。

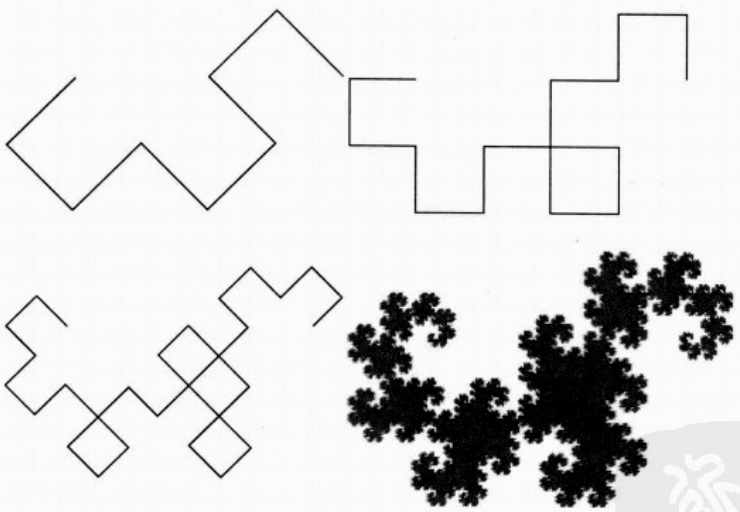


图 4.11　“龙”的形成

例 4.5　希尔宾斯基三角形。

设初始元：R

L 生成元：+R−L−R

R 生成元：−L+R+L

压缩因子：0.5

角度：60°

图 4.12 显示了迭代的头六级运算图，显然其最后图形就是希尔宾斯基三角形。

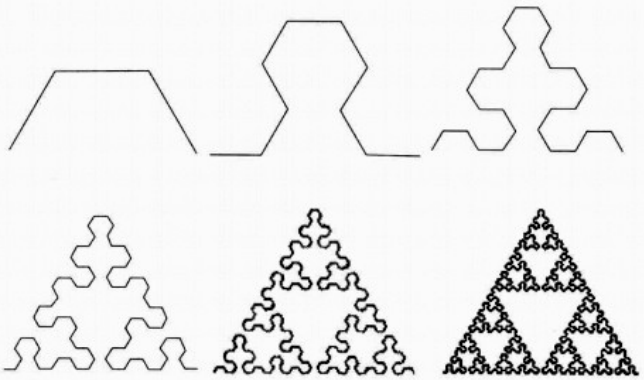


图 4.12　例 4.5 生成的图

4.1.3　分枝结构的简单进退格式

L 系统开始的目的是用于植物结构的绘制工作，应该说，这是 L 系统最重要又最令人感兴趣的一部分。

在前面介绍的两种格式中，它们有一个共同特点，即是在每一级（level）的画图中，总是从左起点开始左拐右拐地一个劲地往前走（中途可以往回退）直到终点为止。也就是说，这种图是终起点各一的图。

但在植物形态中，比如一棵树，它是分枝结构的，即一根树干带大量的分枝，每个分枝都有一个终点，是一种一个起点多个终点的图形。这就意味着在某一级运算中，当画到一个分枝的尽头时画笔必须退回来再画其他结构，这就产生了一种所谓进退操作。这个操作的符号是一对方括号[·]，方括号中通常是三个简单符号，即 F，＋，－．当然方括号中间又可嵌套方括号。当执行完方括号中的指令后，画笔回到左方括号“[”前的位置并保持原先的方向不变。比如 F[＋F]F，F[＋F]－F 和 F[＋F][－F[－F]F]F[＋F][－F]分别表示图 4.13 的三个图形。

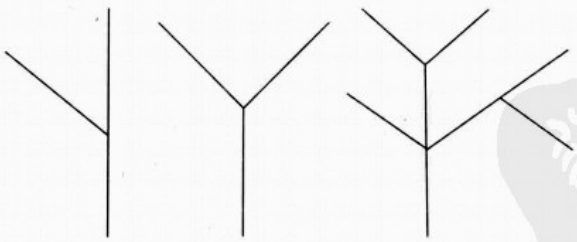


图 4.13　三个分形树的生成元

例 4.6　分形树之一。

初始元：F

生成元：F[－F]F[＋F]F

压缩因子：0.37037

角度 δ：26.5°

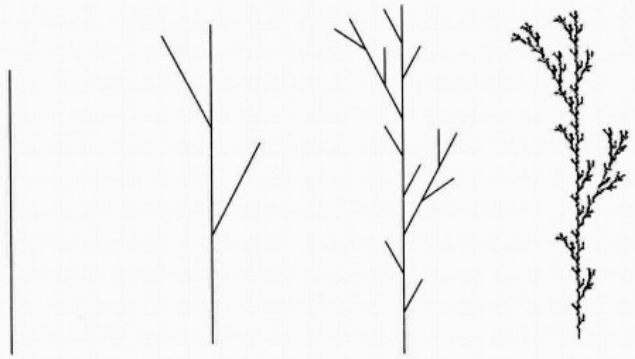


图 4.14　例 4.6 的初始元、生成元和生成的树

生成元的结构中，有右左两个分枝，往上共前进三步，而压缩因子大于 0.33，所以产生的图像会越来越高，仿佛一棵树在生长似的。初始元是 F，故是向上的一条单位线段。头两级应为图 4.14 中的头两个图形。

例 4.7　分形树之二。

设初始元：F；生成元：FF－[－F＋F＋F]＋[＋F－F－F]；压缩因子：0.5；角度 δ：23°。

生成元中，从起点往上两步后，先右后左出现两个分枝，而每个分枝又分别右凸左凸，最后形成风吹动着树模样，见图 4.15。

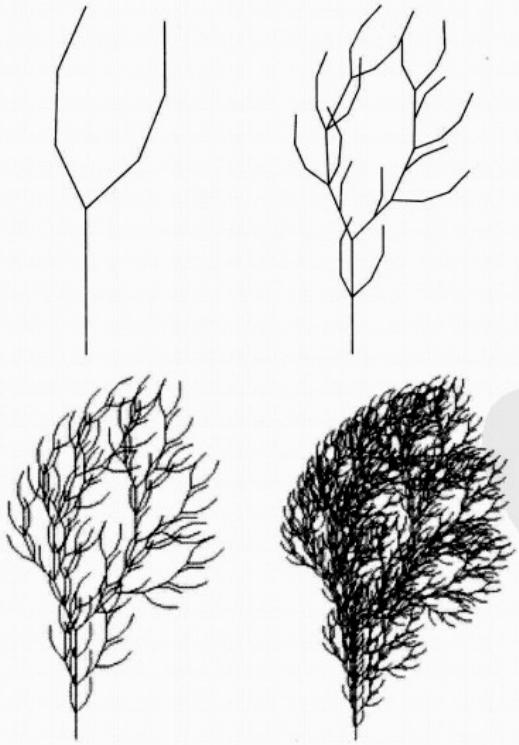


图 4.15　例 4.7 的生成元和生成的树

例 4.8　分形树之三。

初始元：F

生成元：F[＋F]F[－F][F]

角度：20°

压缩因子：0.333

该例与例 4.6 类似，只是最后一个有区别，这里是[F]。生成的图见图 4.16。由于最后符号是[F]，所以画下一个图时，是退回到 F[＋F]F[－F]位置起始画的，这样画出的图比例 4.6 显得枝叶繁茂。

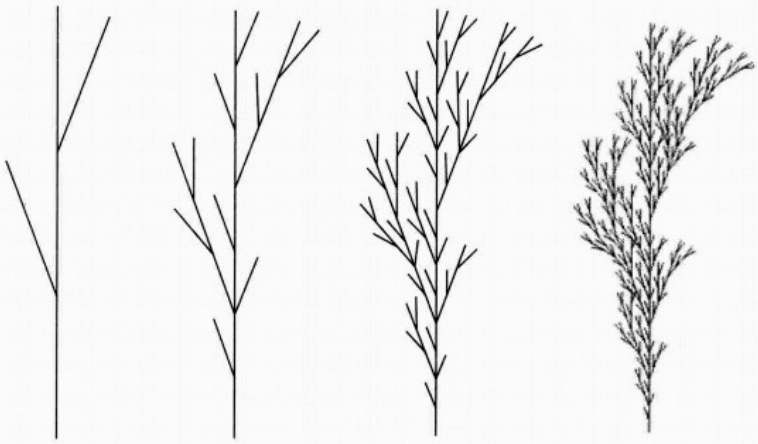


图 4.16　例 4.8 的生成元和生成的树

4.1.4　分枝结构带空指令的进退格式

上一节的分枝结构有个缺点，生成的树从根开始都有分枝，仿如原始森林中一棵从未剪修过的树。我们见到的大多数树不是这样的，它应该是主杆清晰而枝叶繁茂。换言之，在画图时，碰到主杆时就不再分岔而只在分枝中继续分岔不止。为此引进新的称为 X 生成元的做图指令。我们举例说明。

例 4.9　分形树之四。

初始元：X

X 生成元：F[＋X][－X]FX

压缩因子：0.5

角度 δ：27.9°

生成元中的 X 也是个向前符号。在用到的时候，它等同于 F，不用的时候，它就是个空指令。

第一步，初始元是 X，所以它是一条向上的单位线段；

第二步，由 X 生成元生成第二级图。形如图 4.17 中左边第一个小图；

第三步，在走主杆时，X 生成元中实际上成了 FF：压缩了二分之一又走了两步，实际上走完了第一段主杆，接着对两个分枝操作 X 生成元，接着又走第二段主杆，到顶后再操作一次 X 生成元，得左边第二个小图。

重复第三步，直到满意为止，见图 4.17。

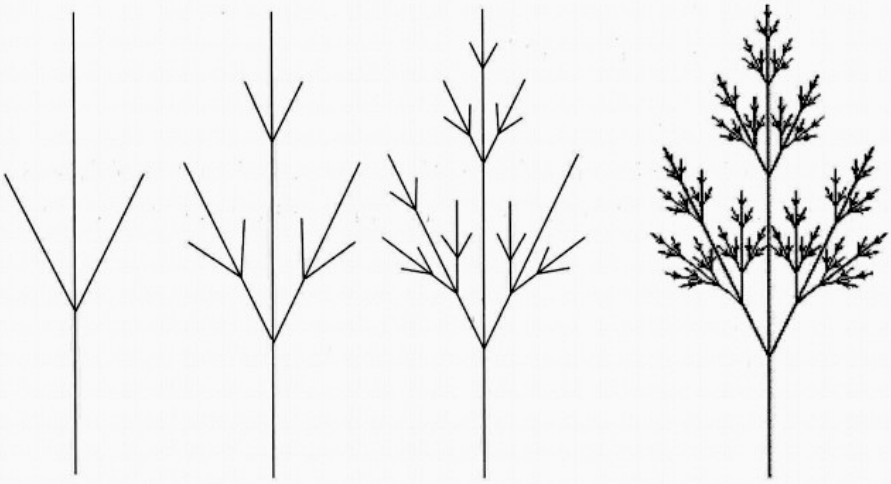


图 4.17　例 4.9 的生成元和生成的树

例 4.10　*分形树之五。*

初始元：X；

生成元：F[＋X]F[－X]＋X；

角度：20°；

压缩因子：0.333。

例 4.10 生成的图见图 4.18。

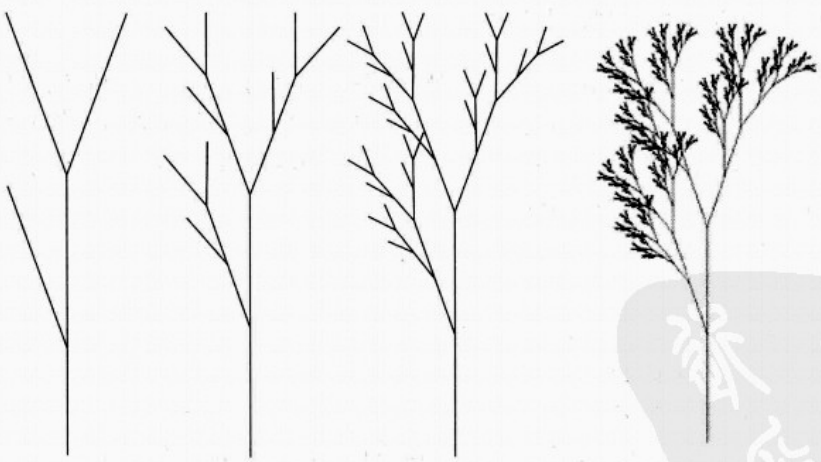


图 4.18　例 4.10 的生成元和生成的树

4.1.5　随机 L-系统

从模拟植物的效果来说，采用前面的方法得到的图形显得不是很自然且有些呆板。我们希望在保留主要特征的前提下，出现一些细节的变化，以求得到生动逼真的图形，于是引进随机操作。

随机性可用在字符解释上（例如 F 中步长 d 可以是随机的），也可以用在生成元上。前者保持基本拓扑结构不变，后者基本拓扑结构则会发生变化，几何性质也发生变化，因此生成的图形也更为“形象化”。

随机 L-系统是有序四元素集 $G=<V, \omega, P, \pi>$。V 为字母表，ω 为公理（初始元），P 是生成元集合，π 是函数：$P \rightarrow (0,1)$，称为概率分布，将生成元集合映射到生成元概率集合上。a 为 V 中任意字符，以它为前驱可以有多个不同的产生方式，所有这些产生方式的概率之和为 1。

在前面非随机 L-系统中，以 a 为前驱的产生式只有一个，于是各个 a 都用相同的后继替代。而在随机系统中产生式不止一个，设为 P_1，P_2，…，P_m，它们的前驱同为 a，后继不同。取生成元 P_i 的概率为 $p_i=\pi(P_i)$，在连接前驱和后继的箭头符号上，写上相应的概率。下面是随机 L-系统的一个例子：

$$\omega: F$$

$$P_1: F \xrightarrow{p_1} \mathrm{F}[+\mathrm{F}]\mathrm{F}[-\mathrm{F}]\mathrm{F}$$

$$P_2: F \xrightarrow{p_2} \mathrm{F}[+\mathrm{F}]\mathrm{F}[-\mathrm{F}[+\mathrm{F}]]$$

$$P_3: F \xrightarrow{p_3} \mathrm{FF}[-\mathrm{F}+\mathrm{F}+\mathrm{F}]+[+\mathrm{F}-\mathrm{F}-\mathrm{F}]$$

其中 $p_1+p_2+p_3=1$。例如取 $p_i=1/3$，$i=1,2,3$。图 4.19 给出了第 6 级时生成的不同分支结构，可以看出这好像是同一种植物的 6 种不同形式。

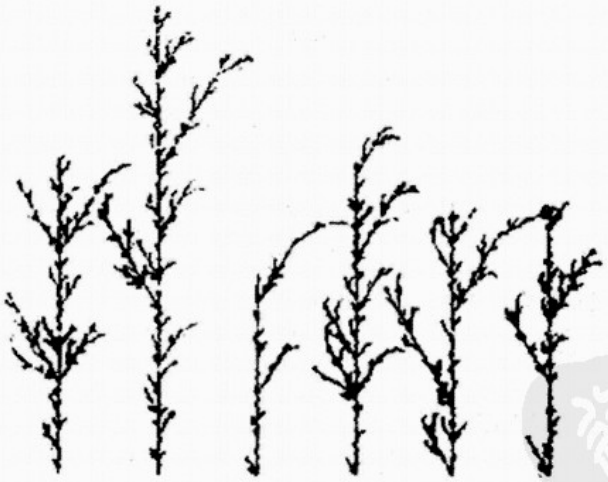


图 4.19　随机分形树

4.2　三维 L-系统

在前面几节，我们介绍了 2D 空间的 L-系统。有人把这种前进（F），左拐（+），右拐（–）的进行方式称为龟行几何。这种方法当然也可以推广到 3D 空间，主要问题是如何表示空间中龟爬行的方向。我们用三个向量 $\boldsymbol{H}$，$\boldsymbol{U}$，$\boldsymbol{L}$ 指示龟的朝向。$\boldsymbol{H}$ 表示向前

(Heading)，$\boldsymbol{U}$ 表示向上（Up），$\boldsymbol{L}$ 表示向左（Left）。这些向量具有单位长度，互相正交，并且满足方程 $\boldsymbol{H}\times\boldsymbol{L}=\boldsymbol{U}$，见图 4.20。

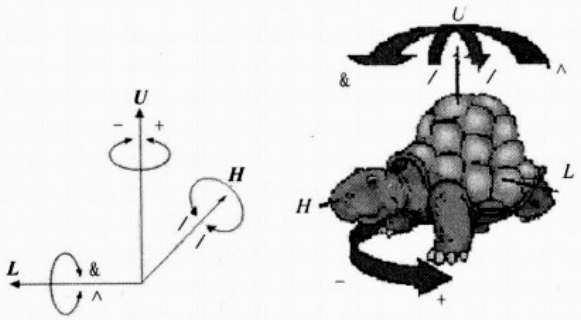


图 4.20　3D 空间的前进方向和旋转

龟的旋转可以表示为

$$[\boldsymbol{H}'\quad \boldsymbol{L}'\quad \boldsymbol{U}']=[\boldsymbol{H}\quad \boldsymbol{L}\quad \boldsymbol{U}]\boldsymbol{R}$$

其中 $\boldsymbol{R}$ 为旋转矩阵。绕 $\boldsymbol{U}$ 逆时针旋转 θ 角的旋转矩阵用 $\boldsymbol{R}_U(\theta)$ 表示；绕 $\boldsymbol{L}$ 及绕 $\boldsymbol{H}$ 的旋转矩阵分别用 $\boldsymbol{R}_L(\theta)$ 和 $\boldsymbol{R}_H(\theta)$ 表示，它们分别有以下形式：

$$\boldsymbol{R}_U(\theta)=\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$\boldsymbol{R}_L(\theta)=\begin{bmatrix}\cos\theta & 0 & -\sin\theta\\ 0 & 1 & 0\\ \sin\theta & 0 & \cos\theta\end{bmatrix}$$

$$\boldsymbol{R}_H(\theta)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\theta & -\sin\theta\\ 0 & \sin\theta & \cos\theta\end{bmatrix}$$

三维系统的命令较多，对三个不同的方向向量，我们采用不同的说法。对朝上向量 $\boldsymbol{U}$，采用左转右转说法；对朝左向量 $\boldsymbol{L}$，用上仰下俯说法；对前进向量 $\boldsymbol{H}$ 则用左滚右滚说法。

下面罗列的是龟在空间改变方向采用的符号：

（1）定位命令　　意义

+	绕朝上向量 $\boldsymbol{U}$ 左转
+(x)	绕朝上向量 $\boldsymbol{U}$ 左转 x 角
–	绕朝上向量 $\boldsymbol{U}$ 右转
–(x)	绕朝上向量 $\boldsymbol{U}$ 右转 x 角
&	绕左向量 $\boldsymbol{L}$ 下俯
&(x)	绕左向量 $\boldsymbol{L}$ 下俯 x
^	绕左向量 $\boldsymbol{L}$ 上仰
^(x)	绕左向量 $\boldsymbol{L}$ 上仰 x 角

<	绕前进向量 $\boldsymbol{H}$ 逆时针向左滚动
<(x)	绕前进向量 $\boldsymbol{H}$ 逆时针向左滚动 x 角
>	绕前进向量 $\boldsymbol{H}$ 顺时针向右滚动
>(x)	绕前进向量 $\boldsymbol{H}$ 顺时针向右滚动 x 角

（2）特殊定位命令　　意义

\|	绕朝上向量 $\boldsymbol{U}$ 旋转 180 度
%	绕前进向量 $\boldsymbol{H}$ 旋转 180 度
$	滚动到水平方向
~	转动/俯仰/滚动
~(x)	在三个方向上按 x 的最大值加进随机方向成分到当前朝向。

（3）移动命令　　意义　　当{ }启动时

F	向前移动且画满全程	记录于顶点
F(x)	向前移动 x 且画满全程	记录于顶点
Z	向前移动且画一半长度	记录于顶点
Z(x)	向前移动 x 且画一半长度	记录于顶点
F	向前移动全程	记录于顶点
f(x)	向前移动 x	记录于顶点
z	向前移动半程	记录于顶点
z(x)	向前移动 x	记录于顶点
g	向前移动全程但不予记录	
g(x)	向前移动 x 但不予记录	
.	不移动但记录于顶点	

（4）结构命令　　意义

[	压进当前状态
]	弹出当前状态
{	开始多边形形状
}	结束多边形形状

注意：建立适用的多边形能在不用移动命令（f，z，g）和定位命令时得到最好的存档。

（5）增减命令　　意义

"	增加长度 10%
"(x)	长度乘以 x
'	减少长度 10%
'(x)	长度乘以 x
;	增加角度 10%
;(x)	角度乘以 x
:	减少角度 10%
?	厚度乘以 1.4
?(x)	厚度乘以 x



!	厚度乘以 0.7
!(*x*)	厚度乘以 *x*

（6）附加命令　　意义

c	增加颜色指标
c(x)	对 x 设立颜色指标
@	文件结束符号
#	注释语句

上面的所有命令都只包含一个字符，并且能够在命令后面加进语句。比如若规定角度为 10 度时，就可用+（30）取代+++。当你还不能确定当前的方向时，可首先用“$”符号，它将使龟行方向落在水平线上，从而可确定当前的 ***U*** 向量朝向 ***Z*** 轴的正方向。

龟行起始于原点，这时候 *X* 轴将作为前进方向 *H*，*Y* 轴为左向量 ***L*** 方向。通过在初始元前面加“&（90）”，龟行轴将与右旋坐标系统一致。

很多数值是自动生成的随机数。应用～（*xx*）将在三个方向上按 *xx* 的最大值加进随机方向成分到当前朝向。

程序 4.1　三维 L-系统基本图形之一。

```
2  # 递归深度(次数)
25  # 角度
50  # 长度的%作为厚度
F   # 画满全程
@   # 文件结束
```

程序 4.2　三维 L-系统基本图形之二。

```
4
25
50
A  # 初始元
A=-FA
@
```

程序 4.3　三维 L-系统基本图形之三。

```
4
25
50
AB
A=-FA
B=+FB
@
```

程序 4.3 的解释：

这个文件（程序）有四个循环，由于 A= –FA，所以有

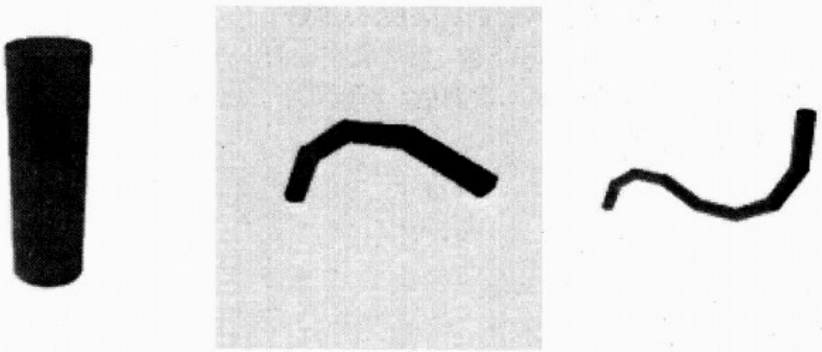
循环 1：–F 结果是 A= –F，

循环 2：–F–F，结果是 A= –F–F，

循环 4：–F–F–F–F，结果是 A= –F–F–F–F。

A= –FA 中，第一个参数是角度 25，接着是画满全程。再次改变角度 25，再次画满全程等等。程序中又有一个正号（+FB），这就意味着是–F–F–F–F+F+F+F+F（=AB）。这最后的字符串将是[–F–F–F–F][+F+F+F+F]。

程序执行的三个图形依次见图 4.21 的(a), (b), (c)。



(a)　　(b)　　(c)

图 4.21　执行程序 4.1 至程序 4.3 的图形

程序 4.4　三维 L-系统基本图形之四。

```
4
25
50
[A]c[B]c[C]c[D]
A=+FA
B=-FB
C=^FC
D=&FD
@
```

程序 4.5　低矮花丛。

```
8
25
50
E
E=[A][B][C][D]
A=c+FAE
B=c-FBE
C=c^FCE
D=c&FDE
@
```

执行它们的图形分别见图 4.22 的(a)和(b)。



(a)　　(b)

图 4.22　执行程序 4.4 和程序 4.5 的分形

程序 4.14 的最后字符串将是

[+F+F+F+F][–F–F–F–F][^F^F^F^F][&F&F&F&F]

“^”和“&”分别绕 L 向量上仰和下俯，这将使在空间里，图形会落在 U 轴的左右两边。“c”是着色命令，每次都要变更一次颜色，所以四个分杈颜色不同（红、蓝、紫、绿）。

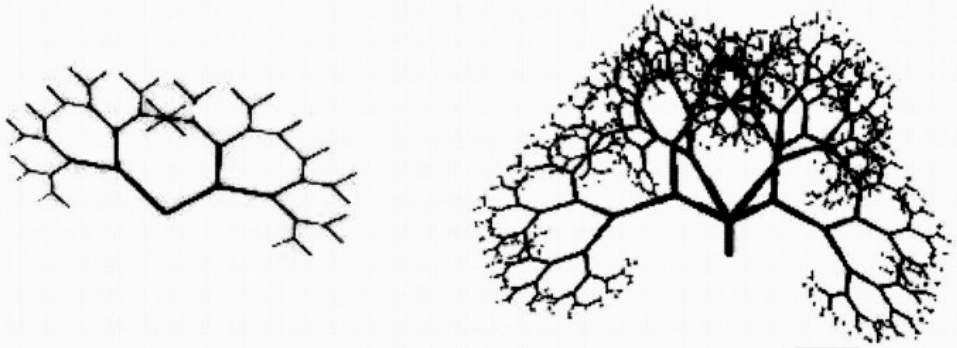
程序 4.6　树枝枝杈。

```
5
45
10
A
A=[+BFA][-BFA]
B='(0.7071)
  @
```

程序 4.7　一棵树。

```
9
45
10
AB
A=[F[+FCA][-FCA]]
B=[F[^FCA][&FCA]]
C='(0.7071)
@
```

这两个程序的图形分别见图 4.23 的(a)和(b)。



(a)　　(b)

图 4.23　程序 4.6 和程序 4.7 生成的分形

程序 4.8　圆球藤筐。

```
18
20
25
#
&(90)A
#
A=[-A]F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F+F
@
```

执行结果见图 4.24。

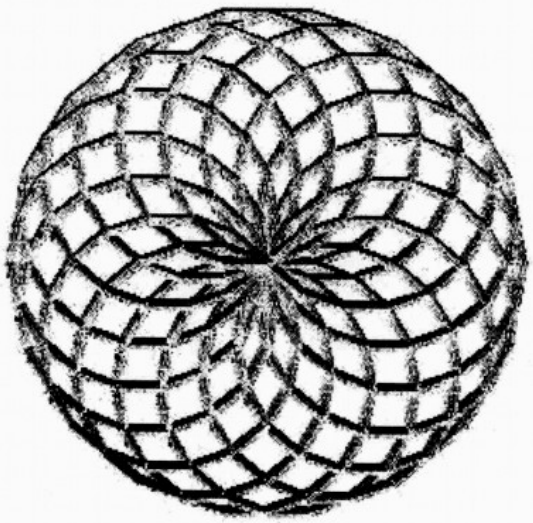


图 4.24　程序 4.8 之分形

程序 4.9　树之俯瞰。

```
6
17
300
Q
Q=X>(45)X>(45)X>(45)X>(45)X>(45)X>(45)X>(45)X
X=[c(2)FFFFFF&(30)c(2)FFAYZ]
Y=[h+B[+EEEEEE[h+CiDiD][h-CiDiD]][h-EEEEEE[h+CiDiD][h-CiDiD]]]
Z=[h-B[+EEEEEE[h+CiDiD][h-CiDiD]][h-EEEEEE[h+CiDiD][h-CiDiD]]]
A=aEA
B=EbEbB
C=aGaGaGaGaGaGatGtGC
D=bHD
a=&(6)
b=^(9)
F=?(0.98)'(0.995)F
h=~(6)
i=?(0.94)i
E=c(1)F[+(40)^(60)?(0.05)'(4)c(8)F][-(40)^(60)?(0.05)'(4)c(8)F]
G=c(1)F[+(40)^(60)?(0.03)'(2)c(8)F][-(40)^(60)?(0.03)'(2)c(8)F]
H=c(1)F[+(40)^(60)?(0.02)'(1)c(8)F][-(40)^(60)?(0.02)'(1)c(8)F]
@
```

得到图形，见图 4.25。

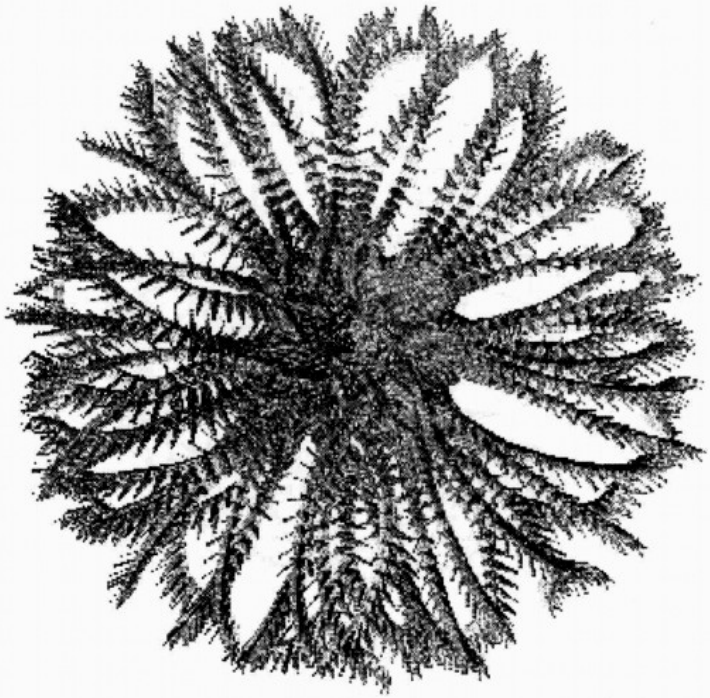


图 4.25　程序 4.9 之分形

程序 4.10　花束。

```
12
5
30
#
c(12)T
#
T=[F~(5)FSd]
S=~(5)FRR[Ba]d>(30)~(5)FRR[Ba]d>(30)!(.8)S
R=[Ba]d>(120)
#
a=~(10)$F[Cxd]~(10)$Fd!(.8)b
b=~(10)$F[Dyd]~(10)$Fd!(.8)a
d=[g(300)g(40)Lg(40)Lg(40)Lg(40)Lg(40)Lg(40)L]
#
B=&B
C=+C
D=-D
#
x=a
y=b
#
F='(1.25)F'(.8)
#
L=[~f(7)c(8){+(30)f(200)-(120)f(200)-(120)f(200)}]
```

```
f=z
z=_
@
```

执行结果见图4.26。

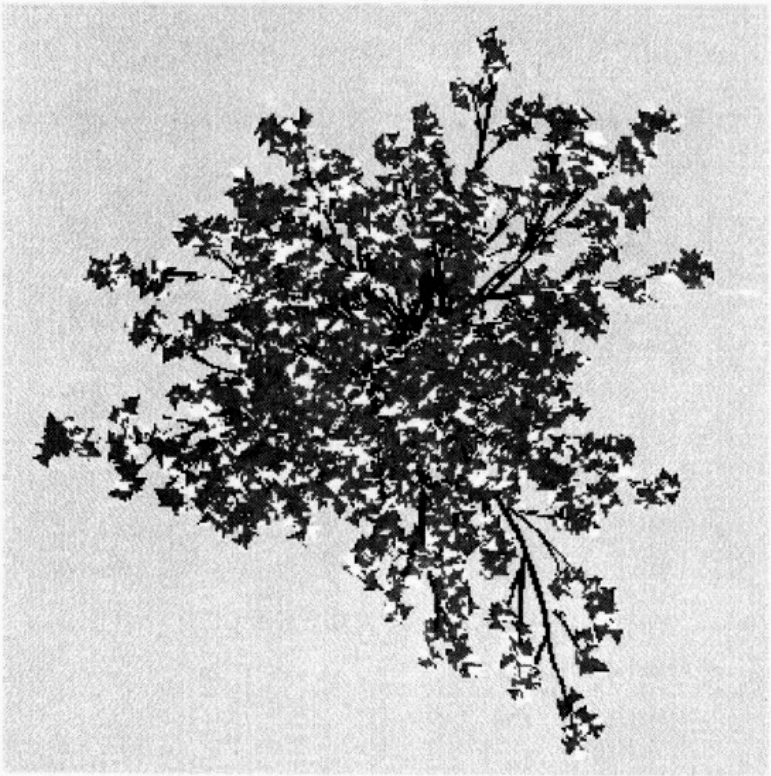


图 4.26　程序 4.10 之分形

程序 4.11　灌木树丛。

```
13
30
50
#
c(12)FFFFFA
#
A=!(.8)tFB>(94)C>(132)D
B=[&'t(.5)!(.9)F$AL|zL]
C=[&'t(.4)!(.8)F$AL|zL]
D=[&'t(.3)!(.7)F$AL|zL]
#
L=[~c(8){+(30)f(40)-(120)f(40)-(120)f(40)}]
@
```

得到图形，见图 4.27。

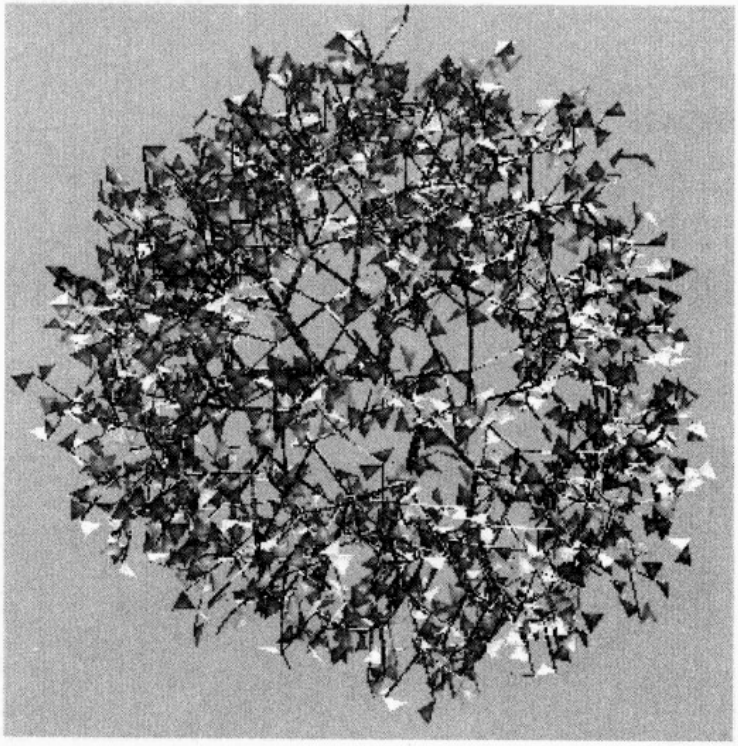


图 4.27　程序 4.11 之分形

程序4. 12　绿叶树。

```
15
20
50
#
c(12)T
#
T=CCA
A=CBD>(94)CBD>(132)BD
B=[&CDCD$A]
D=[g(50)Lg(50)Lg(50)Lg(50)Lg(50)Lg(50)L]
#
C=!(.95)~(5)tF
#
F='(1.25)F'(.8)
#
L=[~f(200)c(8){+(30)f(200)-(120)f(200)-(120)f(200)}]
f=z
z=_
@
```

其分形图见图 4.28。

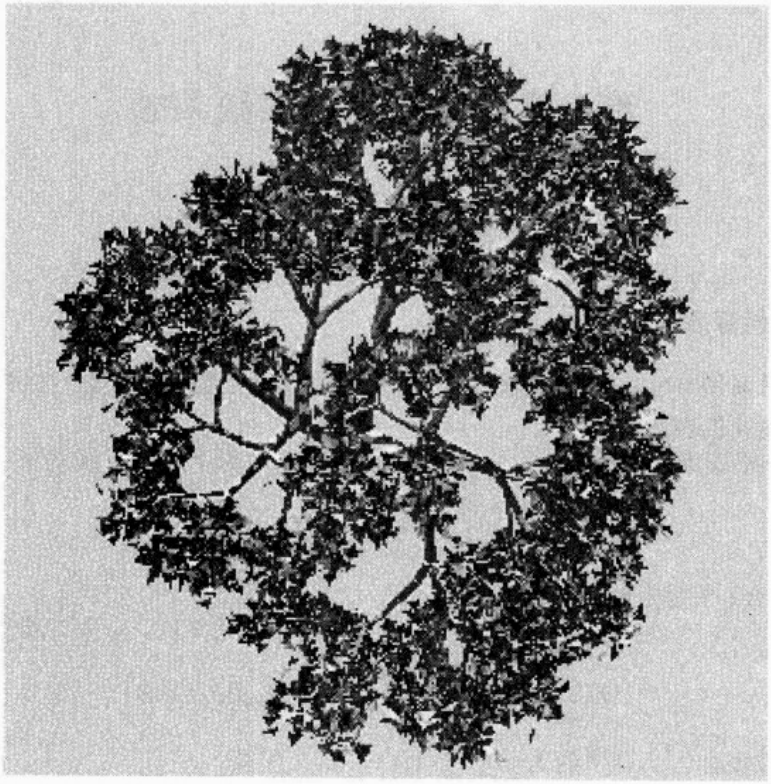


图 4.28　程序 4.12 之分形

第 5 章　迭代函数系统 IFS

5.1　IFS 的算法

5.1.1　确定性算法

迭代函数系统 IFS 的理论和实践在分形学中占有十分重要的地位. 在讲述 IFS 之前，先引进仿射变换的概念。

设 x,y 是平面上的一对变量，a,b,c,d,e,f 是实数。做下面的二维迭代

$$\left.\begin{aligned} x_{n+1} &= ax_n + by_n + e \\ y_{n+1} &= cx_n + dy_n + f \end{aligned}\right\} \quad n = 0,1,2,\cdots \tag{5.1}$$

则式（5.1）就是平面上的一个仿射变换。

可以把式（5.1）写成矩阵变换形式。设 A 为一个二阶矩阵，q 是二维向量，其中

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},\quad q = \begin{bmatrix} e \\ f \end{bmatrix},\quad a,b,c,d,e,f \in R \tag{5.2}$$

式中 R 是实数集合。又设平面上一个点 (x,y)，定义变换

$$W\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{5.3}$$

或等价地写成

$$W(t) = At + q \tag{5.4}$$

式中 $t = [x,y]^{\mathrm{T}}$ 是向量，称变换式（5.3）或式（5.4）为一个（平面上的）二阶仿射变换。

我们说仿射变换 W 是压缩的，如果对平面上两个点 t_1 和 t_2，则有

$$d(W(t_1),\ W(t_2)) \leqslant s \cdot d(t_1,t_2),\quad 0 \leqslant s < 1 \tag{5.5}$$

对某集合 X 中所有不同点 t_1 和 t_2 都成立。式中 $d(t_1,t_2)$ 表示两个点 t_1 和 t_2 之间的距离，s 是其对应的压缩因子。

但仅仅一个仿射变换是不能够画出漂亮的图画来的。设 N 个仿射变换 $W_1, W_2, \cdots, W_N$ 都是压缩的，它们的压缩因子分别是 $s_1, s_2, \cdots, s_N$，定义

$$W(t) = \bigcup_{n=1}^{N} W_n(t) \tag{5.6}$$

则可以证明这个 W 变换也是压缩的，其压缩因子为

$$s = \mathrm{Max}\{s_1, s_2, \cdots, s_N\} < 1$$

以后我们称这个 W 加 s 为一个迭代函数系统 IFS（iterated function system），记之为

$$\{X; W_n, n = 1,2,\cdots,N\} \tag{5.7}$$

具体画图时，仍然是用逃逸时间法，即对给定的集合 E_0，看 n 次迭代后

$$E_n = W^n(E_0),\quad n = 1,2,3,\cdots$$

会产生什么样的效果。如果 W 设计恰当且是压缩的，则

$$S=\lim_{n\to\infty} W^n(E_0)$$

可能会产生出一幅瑰丽的图画来，称 S 是 IFS 的吸引子。

例 5.1　设 E_0 是三个顶点分别在 $A(0,1)$，$B(0,0)$和 $C(1,0)$的一个直角三角形。令

$$W_1\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & 0\\ 0 & \frac{1}{2}\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

$$W_2\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & 0\\ 0 & \frac{1}{2}\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}\frac{1}{2}\\0\end{bmatrix}$$

$$W_3\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & 0\\ 0 & \frac{1}{2}\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}0\\\frac{1}{2}\end{bmatrix}$$

$$W=\bigcup_{n=1}^{3} W_n$$

则 $E=\lim_{n\to\infty} W^n(E_0)$ 是希尔宾斯基三角形。

实际上，由于 W 是连续的，所以只要映射各个顶点，看其变换后落在什么地方，然后连接新的各顶点就行了。

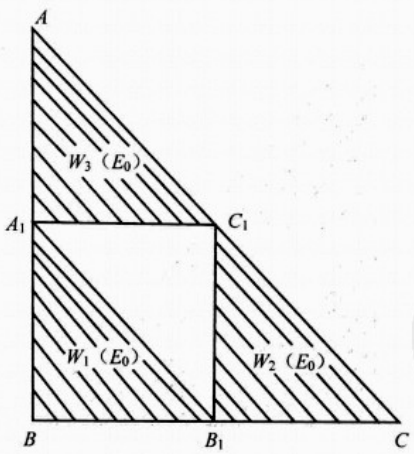


图 5.1　例 5.1W 的第一次映射

显然

$$\left.\begin{aligned}W_1(A)&=A_1\\W_1(B)&=B\\W_1(C)&=B_1\end{aligned}\right\}\Rightarrow W_1(E_0)=\triangle A_1BB_1$$

$$\left.\begin{aligned} W_2(A)&=C_1\\ W_2(B)&=B_1\\ W_2(C)&=C \end{aligned}\right\}\Rightarrow\ W_2(E_0)=\triangle C_1B_1C$$

$$\left.\begin{aligned} W_3(A)&=A\\ W_3(B)&=A_1\\ W_3(C)&=C_1 \end{aligned}\right\}\Rightarrow\ W_3(E_0)=\triangle AA_1C_1$$

则

$$E_1=W_1(E_0)\cup W_2(E_0)\cup W_3(E_0)$$

是图 5.1 中带阴影的图像部分。

迭代的第二步，仅以$W_1(E_1)$为例（见图 5.2）。

$$\left.\begin{aligned} W_1(A_1)&=A_2\\ W_1(B)&=B\\ W_1(B_1)&=B_2 \end{aligned}\right\}\Rightarrow W_1^2(E_0)=\triangle A_2BB_2$$

$$\left.\begin{aligned} W_1(A_1)&=A_2\\ W_1(C_1)&=C_2\\ W_1(A)&=A_1 \end{aligned}\right\}\Rightarrow W_1W_3(E_0)=\triangle A_1A_2C_2$$

$$\left.\begin{aligned} W_1(C_1)&=C_2\\ W_1(B_1)&=B_2\\ W_1(C)&=B_1 \end{aligned}\right\}\Rightarrow W_1W_2(E_0)=\triangle C_2B_2B_1$$

从而可知，该迭代函数系统的吸引子就是希尔宾斯基三角形。

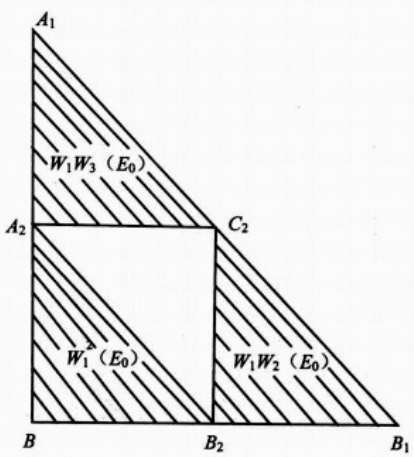


图 5.2　$W_1(E_1)$ 的图示

图 5.3 显示了 E_0 的前五次迭代过程。

图 5.4 是最后的迭代结果。这时吸引子 S 满足

$$S=W(S)=\bigcup_{n=1}^{3}W_n(S)\tag{5.8}$$

我们称满足方程（5.8）的 S 是 W 的不动点。

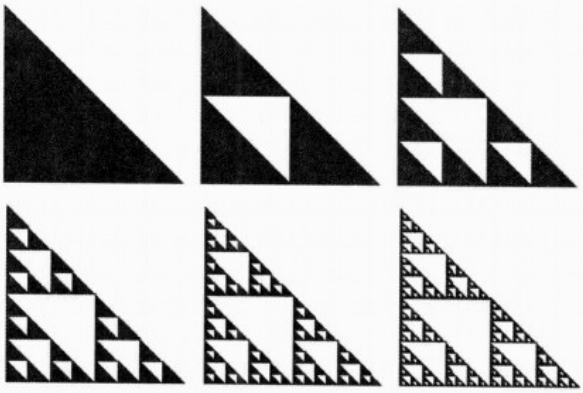


图 5.3　例 5.1 的前五次迭代结果

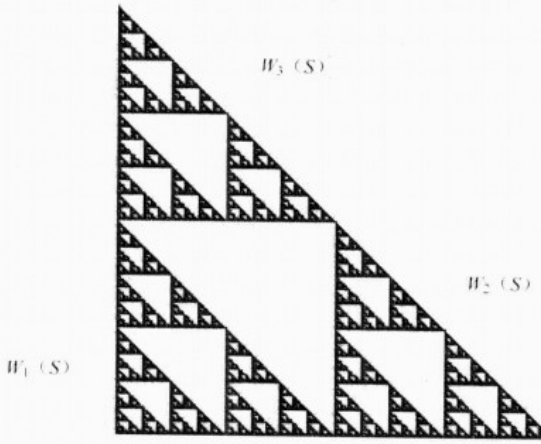


图 5.4　希尔宾斯基三角形是例 5.1 中 W 的不动点

可以这么说，不动点的产生与初始集合（图形）的选择是无关的。例 5.1 中，初始集合 E_0 是充满阴影的直角三角形（见图 5.3），如果选择是正方形或空心的三角形甚至一条线段或其他任何图形，其结果肯定不变，见图 5.5 和图 5.6。

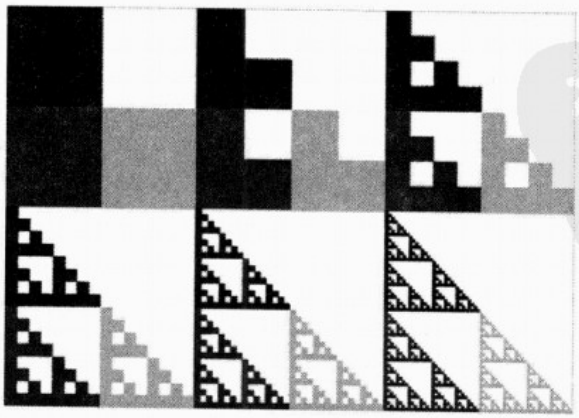


图 5.5　希尔宾斯基三角形的产生过程

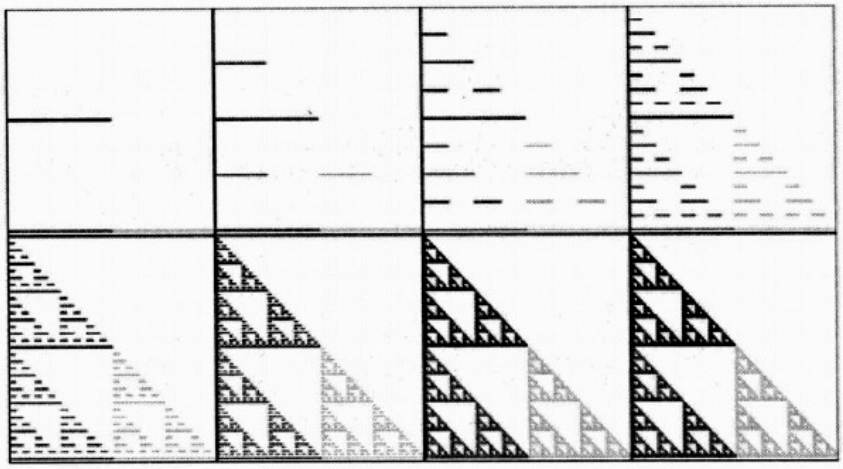


图 5.6　希尔宾斯基三角形的另一产生过程

5.1.2　非确定性算法

前面的仿射变换中，每个变换含有六个参数 a,b,c,d,e,f 。我们称这种算法为确定性算法，该算法有很大的局限性。

例 5.2　设 IFS 为 $\{R^2;W_1,W_2,W_3,W_4\}$，4 个仿射变换的参数由表 5.1 给出，则其吸引子是一张蕨类叶子。图 5.7 显示了该分形的产生过程。

表 5.1　蕨叶的 IFS 码

	a	b	c	d	e	f
W_1	0	0	0	0.17	0	0
W_2	0.8496	0.0255	−0.0255	0.17	0	3
W_3	−0.1554	0.235	0.1958	0.8496	0	1.2
W_4	0.1554	−0.235	0.1958	0.1864	0	3

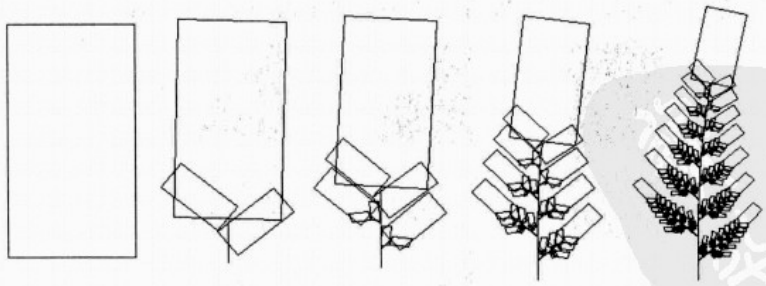


图 5.7　确定性算法蕨叶子产生过程

该例的初始集合 E_0 为一个矩形，每完成一次迭代便要增加 4 倍的矩形，要使一张蕨叶清楚地显示出来，大约必须进行 50 次迭代，即要计算 $\frac{4}{3}\times 4^{50}\approx 1.69\times 10^{30}$ 个矩形，这个计算量是非常可观的。

实际的分形图像是由随机迭代算法给出的，这种算法在原来的 IFS 中增加一组概率数，称为带概率的迭代函数系统（iterated function system with probability，简称 IFSP）。

一个带概率的随机迭代函数系统 IFSP 由一个 IFS $\{X;W_n, n=1,2,\cdots,N\}$ 与一数集 $\{p_n\}_{n=1}^{N}$ 组成，其中

$$\sum_{n=1}^{N} p_n = 1 \text{ 且 } p_n>0\ (n=1,2,\cdots,N) \tag{5.9}$$

而每个概率 p_n 对应于一个变换 W_n，所以一个 IFSP 就是

$$\{X;W_n,p_n,n=1,2,\cdots,N\} \tag{5.10}$$

一般情况下，p_n 由下式给定

$$p_n=\frac{|A_n|}{\sum_{i=1}^{N}|A_i|}=\frac{|a_nd_n-b_nc_n|}{\sum_{i=1}^{N}|a_id_i-b_ic_i|} \tag{5.11}$$

若对某个$|A_n|=0$，则 p_n 可取一小数为 0.001。

具体运算时，可任选一个点为 x_0，然后根据概率产生一个随机整数 $k(1\leqslant k\leqslant N)$，令 $x_1=W_k(x_0)$ 且在 x_1 点处画上一个颜色，然后又根据概率产生另一个随机整数 $j\ (1\leqslant j\leqslant N)$，令 $x_2=W_j(x_1)$，也在该点画上颜色，如此等等。下面是该算法的一个伪码：

$x_0=x$

for n=1 to nmax

　　$x_n=W_k(x_{n-1})$，当随机整数为 k 时

　　draw x_n

next

例 5.3　IFSP 为 $\{R^2;W_1,W_2,W_3,W_4;p_1,p_2,p_3,p_4\}$，它们的参数值由表 5.2 给出，图 5.8 显示了该分形的产生过程，可见，每次迭代结果不再是许多矩形，而是随机落下的点渐显 IFS 的吸引子。

表 5.2　蕨叶的 IFSP 码

	a	b	c	d	e	f	P
W_1	0	0	0	0.16	0	0	0.01
W_2	0.85	0.04	−0.04	0.85	0	1.6	0.85
W_3	0.2	−0.26	0.23	0.22	0	1.6	0.07
W_4	−0.15	0.28	0.46	0.24	0	0.44	0.07



图 5.8　随机迭代法生成的蕨叶图

图 5.5 和图 5.6 是用 VC++编制的程序画的，由于代码太长，不便列出，这里只给出 DOS 下的程序代码，该代码具有很强的通用性。程序后面还附录了两个 IFSP 码的例子，下一节将会给出更多的 IFS 代码。

程序 5.1　IFSP。

```
/***  IFS=PROGRAME TO GENERATE ITERATED FUNCTION SYSTEMS  ***/

#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#define NUM 10
#define Min(x,y)  ((x)<(y)?(x):(y))
#define Max(x,y)  ((x)>(y)?(x):(y))

void image_draw(char finename[13], int color);

main(void)
{
      int color;
      char filename[13];
      clrscr();
      window(20,8,60,12);
      textattr(WHITE|BLUE*16);
      clrscr();
      cprintf("\n ENTER filename:");
      cscanf("%s",filename);
      cprintf("\n Enter picture color(0-14):");
      cscanf("%d",&color);
      image_draw(filename,color);
      getch();
      getch();
      getch();
      cleardevice();
      closegraph();
      exit(0);
}

void image_draw(char *filename, int color)
{
  int i,j,k,no_transformations,temp, xoffset,yoffset,px,py,p_cum[7],flag=0,
      gmode,gd=DETECT;
  float a[NUM][7],f,p_sum=0,xscale,yscale,newx=0,XMin=0,XMax=0,YMax=0,
      YMin=0,x=0,y=0;
  FILE *fin;
  int iter;

 if((fin=fopen(filename,"rb"))==NULL)
  {
     cprintf("\n  File '%s' not found.", filename);
```

```
    exit(0);
}
  fscanf(fin,"%d",&no_transformations);
  for (i=0; i<no_transformations; i++)
     for(j=0;j<7;j++)
     {
     fscanf(fin,"%f",&f);    a[i][j]=f;
     }
  fclose(fin);
  initgraph(&gd,&gmode,"");
  setbkcolor(15);
  for (i=0; i<no_transformations; i++)
  {
   p_sum += a[i][6];    p_cum[i] = p_sum * 32767;
  }
  outtextxy(2,3,"It is rendering");
  /*for(iter=0;iter<100;iter++)*/
  while (!kbhit())
  {
  for (i=0; i<256; i++)
      {    temp = rand();
           for (k=0; k<no_transformations-1; k++)
           {
               if (temp < p_cum[k])
                  break;
           }
           newx = (a[k][0] * x + a[k][1] * y + a[k][4]);
           y =    (a[k][2] * x + a[k][3] * y + a[k][5]);
           x = newx;
           if ((flag == 0) && (i > 15))
           {
               XMax = Max(x,XMax);  XMin = Min(x,XMin);
               YMax = Max(y,YMax);  YMin = Min(y,YMin);
           }
           else
           {
               px = x*xscale + xoffset;
               py = (y*yscale + yoffset);
               if ((px>=0) && (px<639) && (py>=0) && (py<349))
               putpixel (px,349-py,color);
           }
      }
      if (flag == 0)
      {
           xscale = 418 / (XMax - XMin);
           yscale = Min(315 / (YMax - YMin),xscale/1.38);
           if (yscale < xscale/1.38);
               xscale = 1.38 * yscale;
           xoffset = 320 - (XMax+XMin) * xscale / 2;
           yoffset = 175 - (YMax + YMin) * yscale / 2;
```

```
                flag = 1;
            }
        }
        printf("press any key to stop!");
    }

    1. filename: mleaf.ifs
    5
    0.35173 0.35537 -0.35537 0.35173 0.3545 0.5000 0.1773
    0.35338 -0.3537 0.35373 0.35338 0.2879 0.1528 0.38
    0.5 0 0 0.5 0.25 0.462 0.1773
    0.5154 -0.0018 0.00157 0.58795 0.2501 0.1054 0.2091
    0.00364 0 0 0.57832 0.5016 0.0606 0.0562

    2. filename: tree.ifs
    4
    0 0 0 0.5 0 0 0.05
    0.1 0 0 0.1 0 0.2 0.15
    0.42 -0.42 0.42 0.42 0 0.2 0.4
    0.42 0.42 -0.42 0.42 0 0.2 0.4
```

运用 Maple 语言来编 IFS 程序要简单得多。

程序 5.2　IFS。

```
> transforme_point := proc(t, p)
>    [t[1]*p[1]+t[2]*p[2]+t[5], t[3]*p[1]+t[4]*p[2]+t[6]]
> end:

> transforme_triangle := proc(t, triangle)
>    local i;
>    [seq(transforme_point(t, triangle[i]), i=1 .. 3)]
> end:

> IFS := proc(n, liste_de_transformations)
>    local i, j, k, s, sequence_de_triangles:
>    options `Copyright by Alain Schauber, 1996`;
>    sequence_de_triangles := [[0, 0], [1, 0], [0.5, 1]];
>    for j to n do
>    s := NULL;
>    for i to nops(liste_de_transformations) do
>       s := s,
>       seq(transforme_triangle(liste_de_transformations[i],
>       op(k, [sequence_de_triangles])),
>       k=1 .. nops([sequence_de_triangles]))
>     od;
>     sequence_de_triangles := s
>   od;
>   plots[polygonplot]([sequence_de_triangles], axes=none,
>      color=black, scaling=constrained)
> end:
```

编译完上述程序后，执行

```
> rameau:=
> [[.387,.430,.430,-.387,.2560,.5220],
```

```
> [.441,-.091,-.009,-.322,.4219,.5059],
> [-.468,.020,-.113,.015,.4,.4]]:

> IFS(7,rameau);
```

则得到图 5.9。

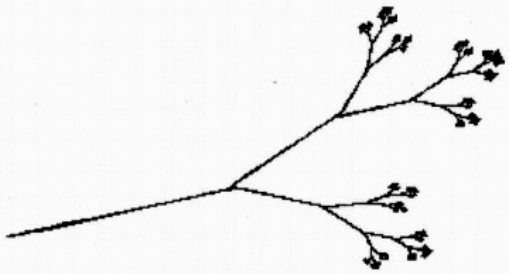


图 5.9　Maple 生成的分形之一

继而执行

```
> arbre_de_Noel_double:=
>[[0,-.5,.5,0,.5,0],[0,.5,-.5,0,.5,.5],[.5,0,0,.5,.25,.5]]:
>
>IFS(7,arbre_de_Noel_double);
```

则画出图 5.10。

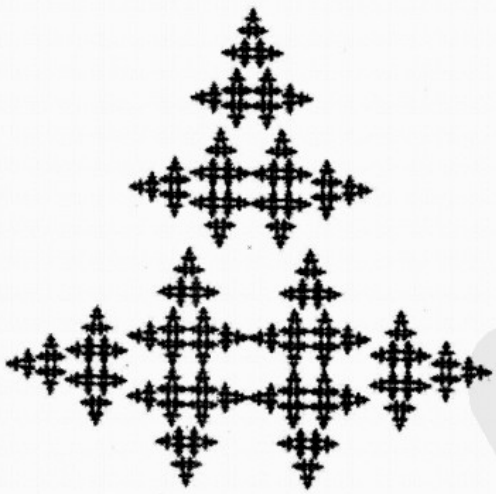


图 5.10　Maple 生成的分形之二

5.2　IFS 的编码

前面两个例子中都是由给定的 IFS 而求得其吸引子。反过来，如何从给出的分形图而得到其 IFS 呢？后者显然是更重要的事。

5.2.1　解线性方程组的编码法

这种方法是找出原像 E_0 和第一次迭代后应得到的 E_1，分析两者的关系后可确定应由几个仿射变换确定该 IFS。比如冯·柯克曲线的原像是线段［0，1］，E_1 由四条折线组成，故产生冯·柯克曲线的仿射变换应有 4 个。其次在 E_0 上确定三个有代表性点的坐标（x_i, y_i），i =1,2,3,并且在 E_1 的相应部位也确定其坐标($\overline{x}_i$, $\overline{y}_i$), i =1,2,3。有了这 6 个参数，就可得到确定仿射变换的 6 个参量 a,b,c,d,e 和 f，具体算法如下：

$$\left.\begin{aligned} x_1a+y_1b+e&=\overline{x}_1\\ x_2a+y_2b+e&=\overline{x}_2\\ x_3a+y_3b+e&=\overline{x}_3 \end{aligned}\right\}\quad 和\quad \left.\begin{aligned} x_1c+y_1d+f&=\overline{y}_1\\ x_2c+y_2d+f&=\overline{y}_2\\ x_3c+y_3d+f&=\overline{y}_3 \end{aligned}\right\} \tag{5.12}$$

解这两个方程组，其中的一个仿射变换就得到了。

5.2.2　旋转和位移变换的编码法

我们将通过举例来介绍如何利用旋转和移动变换来确定仿射变换的 IFS 码。

例 5.4　冯·柯克曲线。

已知冯·柯克曲线的主型如图 5.11 所示，四条折线应有四个仿射变换。第一个变换仅仅是把原线段压缩三分之一而已，因此若令该四个仿射变换为

$$W_n(B)=A_nB+t_n,\qquad n=1,2,3,4$$

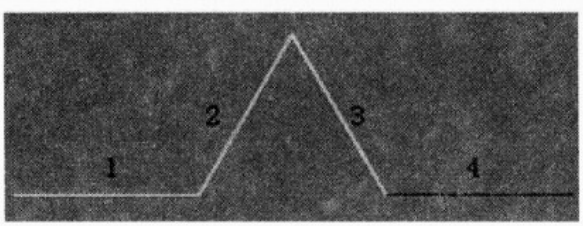


图 5.11　产生冯·柯克曲线的主型

则

$$A_1=\begin{bmatrix}\frac{1}{3} & 0\\ 0 & \frac{1}{3}\end{bmatrix},\qquad t_1=\begin{bmatrix}0\\0\end{bmatrix}$$

W_2 应该把原线段先压缩 1/3，然后左旋 60°，再右移 1/3 即可，即

$$A_2=\frac{1}{3}\begin{bmatrix}\cos 60^\circ & -\sin 60^\circ\\ \sin 60^\circ & \cos 60^\circ\end{bmatrix}=\begin{bmatrix}\frac{1}{6} & -\frac{\sqrt{3}}{6}\\ \frac{\sqrt{3}}{6} & \frac{1}{6}\end{bmatrix},\qquad t_2=\begin{bmatrix}\frac{1}{3}\\0\end{bmatrix}$$

W_3 则是把原线段先压缩 1/3，接着旋转负 60°，随后先右移 1/2，再上移 $\sqrt{3}/6$，即

$$A_3=\frac{1}{3}\begin{bmatrix}\cos(-60°) & -\sin(-60°)\\ \sin(-60°) & \cos(-60°)\end{bmatrix}=\begin{bmatrix}\frac{1}{6} & \frac{\sqrt{3}}{6}\\ -\frac{\sqrt{3}}{6} & \frac{1}{6}\end{bmatrix}$$

$$t_3=\begin{bmatrix}\frac{1}{2}\\ 0\end{bmatrix}+\begin{bmatrix}0\\ \frac{\sqrt{3}}{6}\end{bmatrix}=\begin{bmatrix}\frac{1}{2}\\ \frac{\sqrt{3}}{6}\end{bmatrix}$$

W_4是把原线段先压缩 1/3，然后右移 2/3，即

$$A_4=A_1,\quad t_4=\begin{bmatrix}\frac{2}{3}\\ 0\end{bmatrix}$$

冯·柯克曲线 IFS 码见表 5.3。

表 5.3　冯·柯克曲线 IFS 码

	a	*b*	*c*	*d*	*e*	*f*	*P*
W_1	0.333	0	0	0.333	0	0	0.25
W_2	0.167	−0.289	0.289	0.167	0.333	0	0.25
W_3	0.167	0.289	−0.289	0.167	0.500	0.289	0.25
W_4	0.333	0	0	0.333	0.667	0	0.25

这里要注意的是，对某一条线段的旋转，正反两方向的旋转都可以到达，但这会产生不一样的效果。比如冯·柯克曲线的第三个变换，若是①先压缩 1/3，②左旋 120°，③右移 2/3。这种更简单的方法也能把原单位线段移到图 5.11 的第 3 段位置。如此一来，

$$A_3=\frac{1}{3}\begin{bmatrix}\cos120° & -\sin120°\\ \sin120° & \cos120°\end{bmatrix}=\begin{bmatrix}-\frac{1}{6} & -\frac{\sqrt{3}}{6}\\ \frac{\sqrt{3}}{6} & -\frac{1}{6}\end{bmatrix},\quad t_3=\begin{bmatrix}\frac{2}{3}\\ 0\end{bmatrix}$$

这样的 IFS 会画出下面的图 5.12。



图 5.12　另一种冯·柯克曲线

于是我们有这样的结论：当目标线段下一次迭代后曲线朝线段的左(+)边生长时，应朝左方向(+)旋转；否则旋转向右(−)边进行。冯·柯克曲线主型的第三段下一次生长（从 X 轴上的起始点出发）朝右边，故旋转负角度。

例 5.5　莱维（Levy）曲线。

莱维曲线的主型见图 5.13 左上方小图。后面依此是第 2、3、4 次迭代和最终形成的

图形。

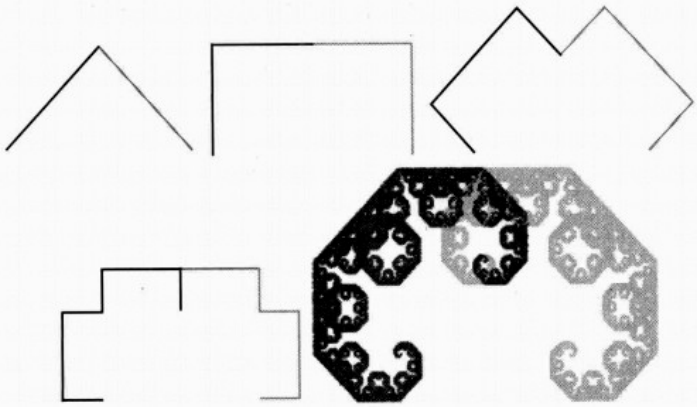


图 5.13　莱维曲线的生成过程

主型由两条折线组成，故应有 2 个仿射变换。令该两个仿射变换为

$$W_n(B)=A_nB+t_n,\qquad n=1,2$$

则 W_1 应该把原线段先压缩 $\frac{\sqrt{2}}{2}$，然后左旋 45°，即

$$A_1=\frac{\sqrt{2}}{2}\begin{bmatrix}\cos 45^\circ & -\sin 45^\circ\\ \sin 45^\circ & \cos 45^\circ\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\\ \frac{1}{2} & \frac{1}{2}\end{bmatrix},\qquad t_1=\begin{bmatrix}0\\0\end{bmatrix}$$

W_2 应该把原线段先压缩 $\frac{\sqrt{2}}{2}$，然后右旋 45，再右移 1/2，上移 1/2 即可，即

$$A_2=\frac{\sqrt{2}}{2}\begin{bmatrix}\cos(-45^\circ) & -\sin(-45^\circ)\\ \sin(-45^\circ) & \cos(-45^\circ)\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\\ -\frac{1}{2} & \frac{1}{2}\end{bmatrix},\qquad t_2=\begin{bmatrix}\frac{1}{2}\\ \frac{1}{2}\end{bmatrix}$$

写成 IFS 码见表 5.4。

表 5.4　莱维曲线的 IFS 码

	a	*b*	*c*	*d*	*e*	*f*
W_1	0.5	−0.5	0.5	0.5	0	0
W_2	0.5	0.5	−0.5	0.5	0.5	0.5

在第 1 章我们曾经给出维数计算公式

$$D=-\frac{\ln N}{\ln r}\tag{5.13}$$

莱维曲线的主型中有两条折线，即 $N=2$，每段长为 $r=2/\sqrt{2}$，所以其分形维数为

$$D=\frac{\ln 2}{\ln\sqrt{2}}=2$$

如果我们把产生 A_2 的旋转换成角度向左转，于是，

$$A_2=\frac{\sqrt{2}}{2}\begin{bmatrix}\cos 145° & -\sin 145°\\ \sin 145° & \cos 145°\end{bmatrix}=\begin{bmatrix}-0.5 & -0.5\\ 0.5 & -0.5\end{bmatrix},\quad t_2=\begin{bmatrix}1\\0\end{bmatrix}$$

它将导致一条龙的产生。图 5.14 左面是前三次迭代结果，这与图 4.10 和图 4.11 的结果完全一致，其 IFS 码见表 5.5。实际上许多 L-系统的图形都可以通过 IFS 产生。

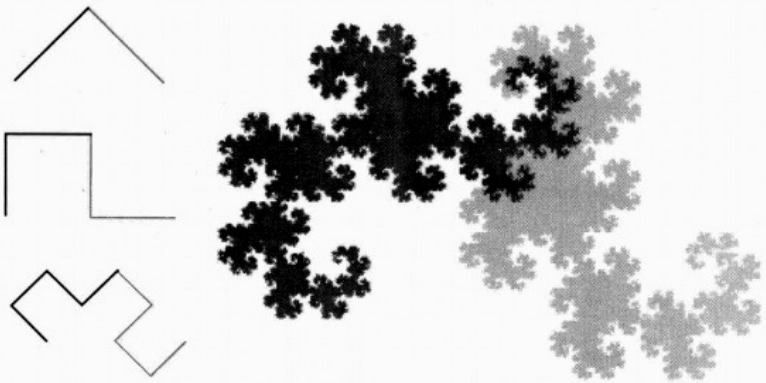


图 5.14　龙曲线的生成

表 5.5　龙曲线的 IFS 码

	a	b	c	d	e	f
W_1	0.5	−0.5	0.5	0.5	0	0
W_2	−0.5	−0.5	0.5	−0.5	1	0

例 5.6　桧树分形小枝。

主型曲线由五条粗黑折线组成，见图 5.15。

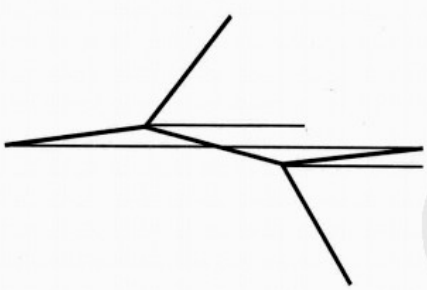


图 5.15　桧树小枝主型

设每条折线长度为 r (>1/3)。第一条和最后一条与水平线的夹角相等，设为 α，则第一条折线与中间那条折线的一半及与水平线构成一个三角形，水平线部分长度为 0.5，由余弦定理，得

$$\frac{3}{4}r-\cos\alpha+\frac{1}{4r}=0 \tag{5.14}$$

令 r =0.34，则 $\alpha=8°=0.13944$。设向上翘起的线段与水平线夹角为 β，β 的角度

当满足$0<\beta<90°$，它与α是独立的。第三条线段与水平线夹角若为θ，则由正弦定理，$\theta=\arcsin(2\sin\alpha)=0.281688=16.14°$。向下的第四条线段与水平线夹角为$(-\delta)$，$\delta$的角度也应当满足$0<\delta<90°$。

我们不妨令$\beta=45^0=0.735898$，$\delta=55°=0.959931$。本 IFS 应有五个变换：

$$W_n(B)=A_nB+t_n,\quad n=1,2,3,4,5。$$

于是

$$A_1=r\begin{bmatrix}\cos\alpha & -\sin\alpha\\ \sin\alpha & \cos\alpha\end{bmatrix},\qquad t_1=\begin{bmatrix}0\\0\end{bmatrix}$$

$$A_2=r\begin{bmatrix}\cos\beta & -\sin\beta\\ \sin\beta & \cos\beta\end{bmatrix},\qquad t_2=\begin{bmatrix}r\cos\alpha\\ r\sin\alpha\end{bmatrix}$$

$$A_3=r\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix},\qquad t_3=\begin{bmatrix}r\cos\alpha\\ r\sin\alpha\end{bmatrix}$$

$$A_4=r\begin{bmatrix}\cos\delta & \sin\delta\\ -\sin\delta & \cos\delta\end{bmatrix},\qquad t_4=\begin{bmatrix}1-r\cos\alpha\\ -r\sin\alpha\end{bmatrix}$$

$$A_5=r\begin{bmatrix}\cos\alpha & -\sin\alpha\\ \sin\alpha & \cos\alpha\end{bmatrix},\qquad t_5=\begin{bmatrix}1-r\cos\alpha\\ -r\sin\alpha\end{bmatrix}$$

得到其 IFS 码如表 5.6 所示。

表 5.6　桧数分形小枝的 IFS 码

	a	*b*	*c*	*d*	*e*	*f*
W_1	0.3367	−0.0473	0.0473	0.3367	0	0
W_2	0.2404	−0.2404	0.2404	0.2404	0.3367	0.0473
W_3	0.3266	0.0945	−0.0945	0.3266	0.3367	0.0473
W_4	0.1950	0.2785	−0.2785	0.1950	0.6633	−0.0473
W_5	0.3367	−0.0473	0.0473	0.3367	0.6633	−0.0473

图 5.16 显示了第 1、2、3 次迭代结果和最后的吸引子。

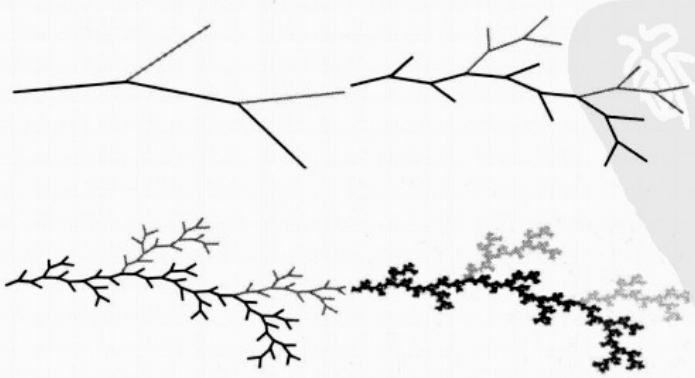


图 5.16　桧树分形小枝

本例中，$N=5$，$r=0.34$，代入公式（5.13），得到桧树分形小枝的分形维 $D=1.492$。

下面将不再推导 IFS 码的建立，只给出 IFS 码表、制图的主型和分形图，见表 5.7 和图 5.17。

表 5.7　花篮族 IFS 码

	a	b	c	d	e	f
W_1	0.5	−0.134	0.134	0.5	0	0
W_2	0.5	0.134	−0.134	0.5	0.5	0.134
W_3	−0.1667	0	0	−0.1667	0.583	0.134
W_4	0.083	−0.229	0.229	0.083	0.417	0.134
W_5	0.083	0.229	−0.229	0.083	0.5	0.359



图 5.17　花篮族

表 5.7 所示 IFS 的 5 个压缩因子并不是都相等的。但根据默兰（Moran）方程

$$\sum_{n=1}^{N} s_n^D = 1 \tag{5.15}$$

再根据前面矩阵 A_n 中

$$r_n \cos\theta = a$$

$$r_n \sin\theta = b$$

即可算出压缩因子 $s_n = r_n$。于是，上例的压缩因子 $s_1 = s_2 = 0.5167$，$s_3 = 0.1667$，$s_4 = s_5 = 0.2435$。最后解方程（5.15）得到 $D = 1.5575$。

表 5.8　花篮 IFS 码

	a	b	c	d	e	f
W_1	0.5	−0.182	0.182	0.5	0	0
W_2	0.5	0.182	−0.182	0.5	0.5	0.182
W_3	−0.1667	0	0	−0.1667	0.5833	0.182
W_4	0.0833	−0.4727	0.4727	0.083	0.417	0.182
W_5	0.083	0.4727	−0.4727	0.083	0.5	0.6547

根据表 5.8 所示的 IFS 码画出的图形见图 5.18。

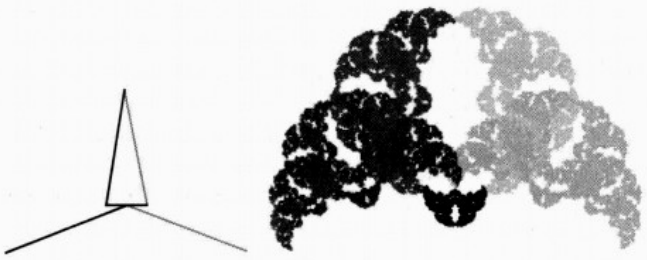


图 5.18　插满鲜花的小花篮

它的分形维 D=2.0755。表 5.9 是五角繁星 IFS 码。

表 5.9　五角繁星 IFS 码

	a	b	c	d	e	f
W_1	0.382	0	0	0.382	0.3072	0.619
W_2	0.382	0	0	0.382	0.6033	0.4044
W_3	0.382	0	0	0.382	0.0139	0.4044
W_4	0.382	0	0	0.382	0.1253	0.0595
W_5	0.382	0	0	0.382	0.492	0.0595

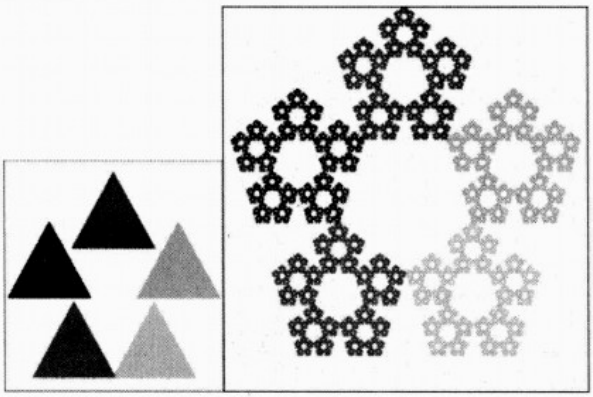


图 5.19　五角繁星

显然图 5.19 的压缩因子为 0.382，其分维数为 $D=1.6724$。

5.2.3　利用拼贴方法的 IFS 编码

本例以枫叶 IFS 的构造说明拼贴方法的应用。

给定的目标图形 L 可以是数字化的图像（如黑色背景上的一片白叶），也可以是近似的多边形（如多边形的树叶边界），如图 5.20（0）所示，括号中的 0 表示初始图像，其他括号中的数字表示迭代的次数，α表示吸引子。现引进平面仿射变换：

$$W_1\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}a_1 & b_1\\c_1 & d_1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}e_1\\f_1\end{bmatrix}$$

一开始可用一个很粗糙的图形去拼贴目标图像的一部分，比如初始化的结果为 $a_1=d_1=1/4,\ b_1=c_1=e_1=f_1=0$。将 W_1 作用于 L 得 $W_1(L)$，它是 L 的四分之一大小的复制品，而且其中心很靠近(0,0)点。现不断调整 W_1 的变换系数 a_1,b_1,c_1,d_1,e_1 和 f_1，使得 $W_1(L)$ 覆盖 L 的大部分（见图 5.20（0）中下方粗分线包围部分），得 $a_1=d_1=0.6,\ b_1=c_1=0$，$e_1=0.18, f_1=0.12$，这样 $W_1(L)$ 就成为 L 图的一个部分。重要的是 $W_1(L)$ 要比 L 小以便 W_1 是压缩的。一旦 $W_1(L)$ 的位置被 L 确定，就引进新的仿射变换 W_2，适当调整 W_2 的变换系数直至 $W_2(L)$ 能覆盖 L 的另一部分（与图形 $W_1(L)$ 不同）。$W_2(L)$ 与 $W_1(L)$ 之间容许有重叠，但这种重叠越少越好。这里我们设定的 $W_2(L)$，实际上是 $W_1(L)$ 往上移动了一段距离，从而 $a_2=d_2=0.6,\ b_2=c_2=0,\ e_2=0.18, f_2=0.38$。进而再引进 W_3，它可以是原图像压缩 0.5 倍后左旋 36.87^0，水平线方向右移 0.27，垂直方向上移 0.36。W_4 是 W_3 的对称反方向旋转，只是上移距离有差别。如此一来，一组压缩仿射变换 $\{W_1,W_2,\cdots,W_N\}$ 就确定了。它具有这样的性质，只要给定的目标图形 L 与集合

$$\tilde{L}=\bigcup_{n=1}^{N}W_n(L)$$

“看上去十分接近”，且 N 要尽量地小，则用这组压缩映射求得的吸引子 A 也“看上去十分接近”于原像 L。

图 5.20（0）显示了枫叶的拼贴过程，求得的 4 个仿射变换的系数如表 5.10 所示，图 5.20（∞）就是其吸引子。

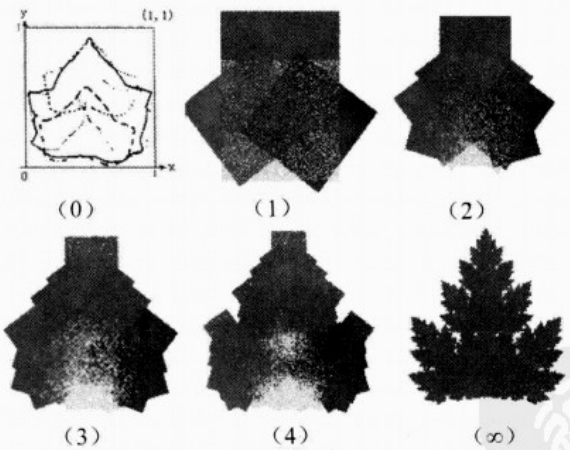


图 5.20　枫叶的拼贴及其 IFS 迭代

表 5.10　枫叶 IFS 码

	a	b	c	d	e	f
W_1	0.6	0	0	0.6	0.18	0.12
W_2	0.6	0	0	0.6	0.18	0.36
W_3	0.4	0.3	−0.3	0.4	0.27	0.36
W_4	0.4	−0.3	0.3	0.4	0.27	0.09

容易算出压缩因子 $s_1 = s_2 = 0.6, s_3 = s_4 = 0.5$。代入默兰公式（5.12），算出该枫树叶子的分维数为 2.3403。

“拼贴”是对于目标图像设计 IFS 码的重要方法.下面给出几个通过“拼贴”得到的 IFS 码及其生成的图像。对应的 IFS 码中如无概率数 p，则表明其概率数相等。表 5.11 所对应的图形见图 5.21。

表 5.11　拼贴分形之一 IFS 码

	a	b	c	d	e	f
W_1	0.195	−0.488	0.344	0.443	0.4431	0.2452
W_2	0.462	0.414	−0.252	0.361	0.2511	0.5692
W_3	−0.058	−0.07	0.453	−0.111	0.5976	0.0969
W_4	−0.035	0.07	−0.469	−0.022	0.4884	0.5069
W_5	−0.637	0	0	0.501	0.8562	0.2513



图 5.21　拼贴分形之一

表 5.12 所对应的图形见图 5.22。

表 5.12　拼贴分形之二 IFS 码

	a	b	c	d	e	f
W_1	−0.04	0	−0.23	−0.65	−0.08	0.26
W_2	0.61	0	0	0.31	0.07	2.5
W_3	−0.65	−0.29	−0.3	0.48	0.54	0.39
W_4	0.64	−0.3	0.16	0.56	−0.56	0.4

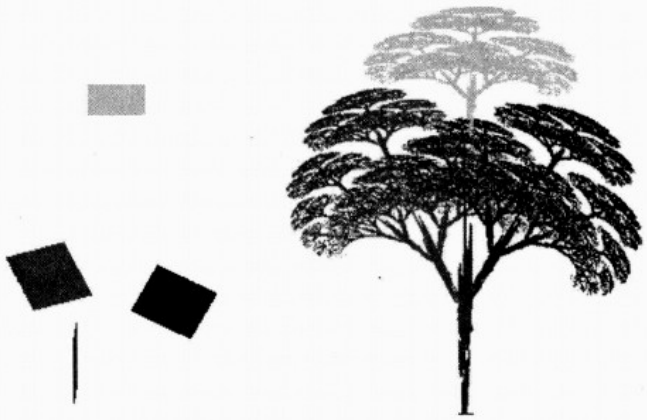


图 5.22　拼贴分形之二

表 5.13 所对应的图形见图 5.23。

表 5.13　拼贴分形之三 IFS 码

	a	*b*	*c*	*d*	*e*	*f*	*p*
W_1	0	0	0	0.6	0.0275	−0.4231	0.043
W_2	0.3636	0.5818	−0.4545	0.4727	−0.1538	3.3893	0.4758
W_3	−0.3636	−0.5818	−0.4545	0.4727	0.2090	3.3893	0.4758

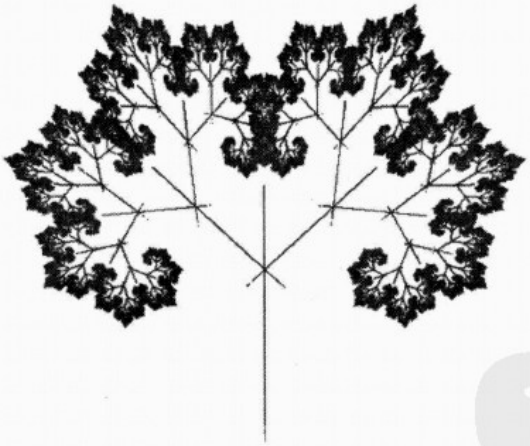


图 5.23　拼贴分形之三

5.3　带参量的 IFS

在前面两节中，我们得到的都是静态的分形图像。对于实时动画，则可以用带参量的 IFS 来处理。

何谓带参量的 IFS？就是在仿射变换的矩阵或（和）向量元素里，加进变量成分。理论上可以证明，只要给予变量在一定范围内以使仿射变换是压缩的，则这个 IFS 的吸

引子就连续地随变量而变化。我们举几个例子来说明。

例 5.7　带变量的希尔宾斯基三角形。

令例 5.1 中的 W_1, W_2 不变，改动 W_3 为

$$W_3\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}\frac{1}{2} & 0\\ 0 & \frac{1}{2}\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}\frac{p}{2}\\ \frac{1}{2}\end{bmatrix},\qquad p\in[0,1]$$

则这个 IFS 的吸引子将随 p 的变动而变化。图 5.24 是 $p=i/7$（$i=0,\cdots,7$）时的八幅图像。

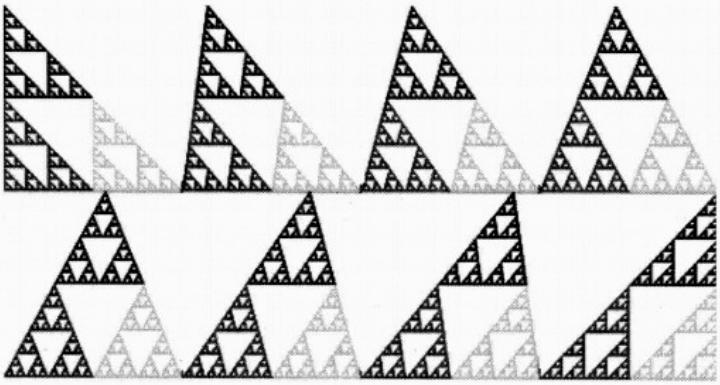


图 5.24　变化中的希尔宾斯基三角形

例 5.8　带参量的蕨叶子。

回看表 5.2，其中 W_2 是产生大量叶子的主要变换。实际上它是一个旋转变换，由于

$$r\cos\alpha=0.85 \text{ 和 } r\sin\alpha=-0.04$$

所以 $r=0.850941$，$\alpha=0.047024$。现在我们令

$$W_2\begin{bmatrix}x\\y\end{bmatrix}=r\begin{bmatrix}\cos k\alpha & -\sin k\alpha\\ \sin k\alpha & \cos k\alpha\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}0\\1.6\end{bmatrix}$$

其中 $r=0.850941$，$\alpha=0.047024$，$k\in[-2,2]$，当我们分别取 $k=-2.0$，-1.5，-1.0，-0.2，0.2，1.0，1.5 和 2.0 时，分别产生的图像见图 5.25。



图 5.25　随风摇动的蕨叶

例 5.9　风中摇曳的树。

令 IFS $\{R^2;W_1,W_2,W_3,W_4\}$ 生成一棵树，IFS 码中，a,b,c,d 改为 rx,ry,θ,ϕ 而 e,f,p 不动。为使得到一棵风中摇曳的树，加风力因子 $powf$，作为参量，它对树的各分支影响因子为（1.0,0.45,0.45,0.10），即对上面分支影响大，对树干影响小，这符合实际情况。计算中，令

$$powf_1 = powf,\ powf_2 = powf_3 = 0.45\ powf,\quad powf_4 = 0.1\ powf$$

$$\left.\begin{aligned} a_n &= rx_n \cos((\theta_n + powf_n)\pi/180) \\ b_n &= -ry_n \sin((\varphi_n + powf_n)\pi/180) \\ c_n &= rx_n \sin((\theta_n + powf_n)\pi/180) \\ d_n &= ry_n \cos((\varphi_n + powf_n)\pi/180) \end{aligned}\right\},\quad n=1,2,3,4$$

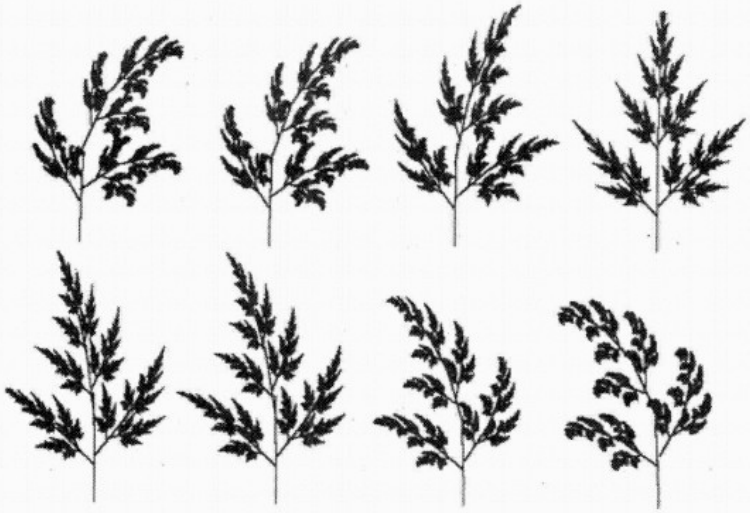


图 5.26　不同参量生成风中摇动的树

由随机迭代算法。可得风中吹歪的树。图 5.26 显示了不同 $powf$ 的八棵树，可以看出，当风力过于大时，树干会被吹断。程序的数据文件由表 5.14 给出。

表 5.14　风中树的 IFS 码

rx	ry	θ	φ	e	f	P
0.52	0.7143	0.00	0.00	0.0	0.60	0.25
0.28	0.5143	45.00	45.00	0.0	0.28	0.25
0.40	0.5143	-45.00	-45.00	0.0	0.35	0.25
0.00	0.4280	0.00	0.00	0.0	0.00	0.25

5.4　三维 IFS

在二维 IFS 码的基础上，加上三维绘图的技巧，包括三维 IFS 码，就会画出一幅生

动的立体图像来。这时候 IFS 的变换矩阵和向量是三阶的，其一般形式是

$$W\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}a & b & c\\d & e & f\\g & h & i\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}+\begin{bmatrix}j\\k\\l\end{bmatrix} \tag{5.16}$$

以生成立体的希尔宾斯基三角形为例，它应有五个仿射变换，具体是

$$W_1\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}0.5 & 0 & 0\\0 & 0.5 & 0\\0 & 0 & 0.5\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}+\begin{pmatrix}0\\0\\0\end{pmatrix}$$

$$W_2\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}0.5 & 0 & 0\\0 & 0.5 & 0\\0 & 0 & 0.5\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}+\begin{pmatrix}0.5\\0\\0.5\end{pmatrix}$$

$$W_3\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}0.5 & 0 & 0\\0 & 0.5 & 0\\0 & 0 & 0.5\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}+\begin{pmatrix}-0.5\\0\\0.5\end{pmatrix}$$

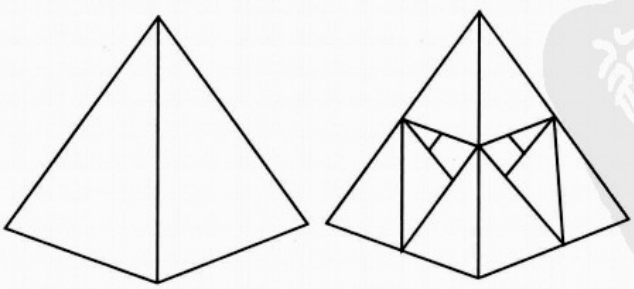
$$W_4\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}0.5 & 0 & 0\\0 & 0.5 & 0\\0 & 0 & 0.5\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}+\begin{pmatrix}0.5\\0\\-0.5\end{pmatrix}$$

$$W_5\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}0.5 & 0 & 0\\0 & 0.5 & 0\\0 & 0 & 0.5\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}+\begin{pmatrix}-0.5\\0\\-0.5\end{pmatrix}$$

其中 z 是图像的厚度。对应图像的初始元和生成元分别见图 5.27 的（a）和（b）。最后的吸引子见图 5.28。

生成三维希尔宾斯基三角体有五个仿射变换，每个压缩因子都为 0.5，所以其分维数为

$$\ln 5 \big/ \ln 2 = 2.3219$$



（a）　　（b）

图 5.27　三维希尔宾斯基三角体的初始元和生成元

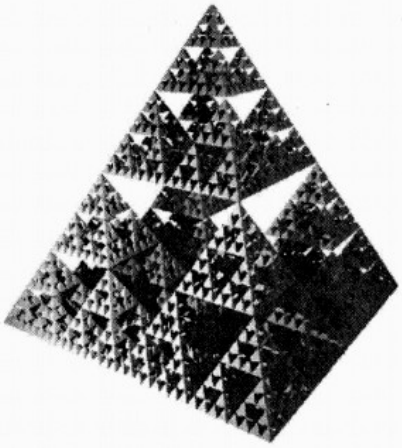


图 5.28　三维希尔宾斯基三角体之一

5.4.1　无厚度的三维图像生成

有些图像如树叶之类，看上去有立体感，但其厚度几乎可以忽略。如果不用厚度变量 z，则其 IFS 码的三阶矩阵和向量是

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{bmatrix} 和 \begin{bmatrix} e \\ f \\ 0 \end{bmatrix} \tag{5.17}$$

其中 a,b,c,d,e,f 是对应的平面仿射变换的六个元素，但这时候要用其他技术才能画出三维图像来。下面的例子中，我们把二维 IFS 码中的数据填入到软件 IFS Builder 3D 的数据文件里。运行软件后可生成三维图像。

例 5.10　蕨叶。

生成三维蕨叶的数据文件是这样的：

```
camera  position (0, 0, -80) vertical(-1, 0, 0)
light   color (0,1,0) position (1,1,-3) shadows(0);
light   color (1,1,0) position (2,6,-3) shadows(0);
ambient(1, 1, 1,.05)
background(0,1,1,1)

f1 := matrix(0,0,0,0,.16,0, 0,0,0);
f2 := translate(0,1.6,0)*matrix(.85,.04,0, -.04,.85,0, 0,0,0);
f3 := translate(0,1.6,0)*matrix(.2,-.26,0,  .23,.22,0, 0,0,0);
f4 := translate(0,.44,0)*matrix(-.15,.28,0, .26,.24,0, 0,0,0);
build Fern = (f1+f2+f3+f4)Fern;
```

生成的图像见彩色图片 5（左）。

下面的术语解释中，坐标（x,y,z）的方向如图 5.29 所示。

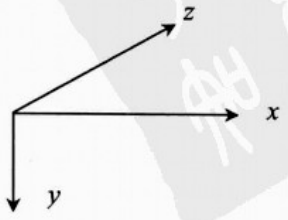


图 5.29　坐标图

camera position（x,y,z）如果把要绘制的图像看成被摄影的对象，则该函数表示摄影机（观察者）的位置。

vertical（x,y,z），表示摄像机的朝向。

程序还采用光照技术 light，它表示在位置点（position）着色（color），还有阴影（shadows）。从图上看，光照效果非常明显。

Ambient（环境光，即来自周围环境没有固定方向的光）函数有四个颜色，依次为浅蓝、浅红、浅黄和灰黑色。这里的意思是往深处的颜色呈深灰色。

Background（背景）函数的四个颜色含义与 Ambient 相同，这里表示此图的背景色是浅蓝。

f1，f2，f3 和 f4 对应二维 IFS 码的四个仿射变换，其中 translate（x,y,z）是对向量的平移，如无位移则可省去该函数。matrix 是对应的变换矩阵。可以看出，它采用了二维仿射变换的数据，只在向量上有些改动。

例 5.11　枫叶。

类似例 5.10，生成三维枫叶的数据文件是：

```
camera  position (0,0,-80) vertical(0,1,0)
light  color (0,1,0) position (1, 1,-3) shadows(0);
light  color (1,1,0) position (3,-1,-3) shadows(0);
ambient(1,1,1, .05)
background(0,1,1,1)
f1 := translate(-0.08,-1.31,0)*matrix(0.14, 0.01,0,0,0.51, 0,0,0,0);
f2 := translate(1.49,-0.75, 0)*matrix(0.43, 0.52,0, -0.45,0.5,
        0, 0,0,0);
f3 := translate(-1.62,-0.74,0)*matrix(0.45,-0.49,0,  0.47,0.47,
        0, 0,0,0);
f4 := translate(0.02,1.62,0)*matrix(0.49, 0,0, 0,0.51,0, 0,0,0);
build Leaf = (f1+f2+f3+f4)Leaf;
```

生成的图像见彩色图片 5（右）。

这里的 IFS 码与二维情况有所不同，画的图像却更逼真。

5.4.2　三维图像的 IFS 生成

这里的立体有厚度，这厚度指的是图 5.29 中 Z 轴方向的深度。

例 5.12　带厚度的希尔宾斯基三角形。

数据文件如下：

```
camera  position (0,0,-80) vertical(1,0,0)
light   color (0,1,0) position (1, 1,-3)
light   color (1,1,0) position (3,-1,-1)
light   color (1,0,0,.02) position (0, 0, 0)
ambient(1,1,1,.05)
background(0,1,1,1)

t=0;//parameter

k1=1/2-t;
k2=1/2+(2*t);
f=0;
```

```
a=sqrt(2)/3;
b=sqrt(6)/3;
f0 :=translate(0,0,1)*rotate(0,0,1,f,k1,k2)*translate(0,0,-1);
f1 :=translate(2*a,0,-1/3)*rotate(2*a,0,-1/3,f,k1,k2)
    *translate(-2*a,0,1/3);
f2 :=translate(-a,b,-1/3)*rotate(-a,b,-1/3,f,k1,k2)
    *translate (a,-b,1/3);
f3 :=translate(-a,-b,-1/3)*rotate(-a,-b,-1/3,f,k1,k2)
    *translate (a,b,1/3);

build Sierpinski=(f0+f1+f2+f3)Sierpinski;

//another
k1=3/5-t;
k2=3/5+(4*t);
f=180;
```

生成的图像见图 5.30。

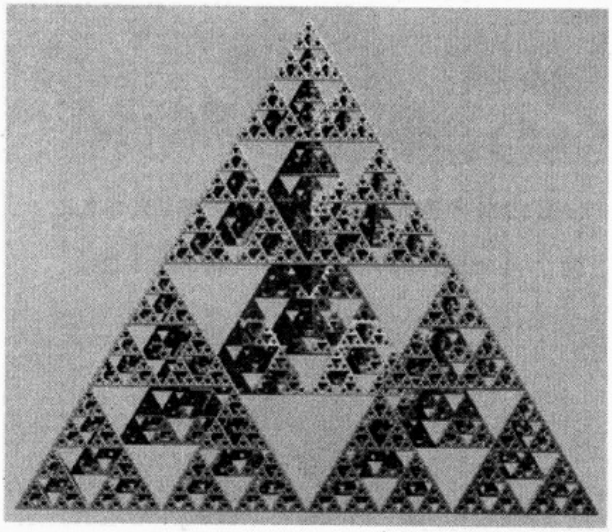


图 5.30　三维希尔宾斯基三角体之二

看得出，这里的加厚是对 Z 轴透视方向而言的。由于真正的三维希尔宾斯基三角体图像像一个金字塔（见图 5.28），所以这里朝 Z 轴方向看过去是一个又一个更小的希尔宾斯基三角形。

与无厚度的三维图不同，这里有三个点光源。

我们在讲二维 IFS 码时，用到了对线段的压缩、旋转和平移。这里也用到了旋转和平移两种变换。旋转函数及其变量是 rotate（x,y,z,angle,along,across），前四个参数表示以从（0,0,0）到（x,y,z）做成的向量为轴（axis），然后旋转 angle 角度。但需沿着该轴要作因子 along 的压缩并且在正交于该轴的平面上做因子 across 的压缩。

此例实际上是带参量的 IFS，图 5.30 是 t=0 的图像。再分别取 t=0.2，0.6，1.0，得到图 5.31。

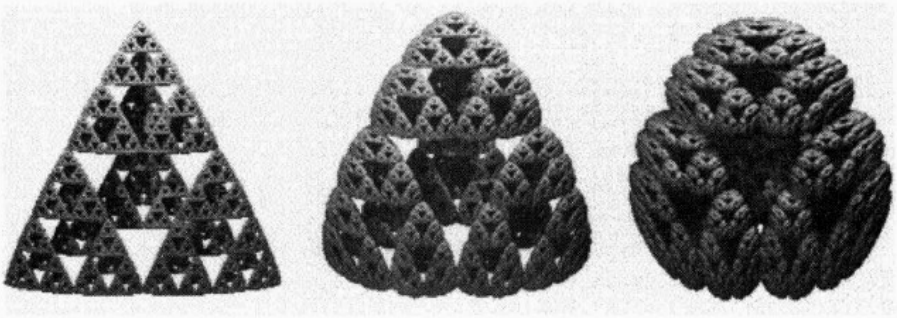


图 5.31　不同参数的三维希尔宾斯基三角体之一

再取 t=–0.02，–0.04，–0.06，生成图像见图 5.32。

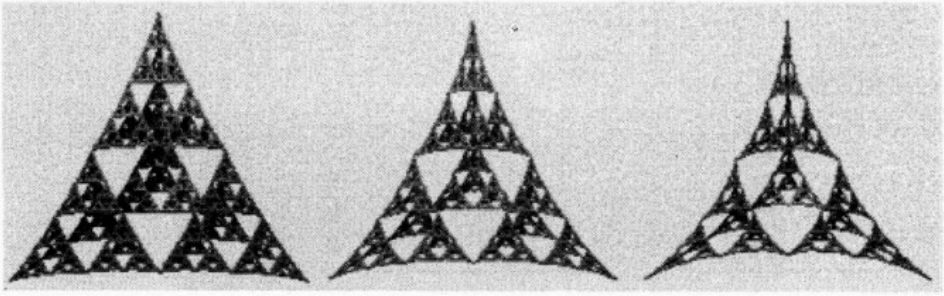


图 5.32　不同参数的三维希尔宾斯基三角体之二

再取数据文件中的“//another”参数（t=0），得到图 5.33。

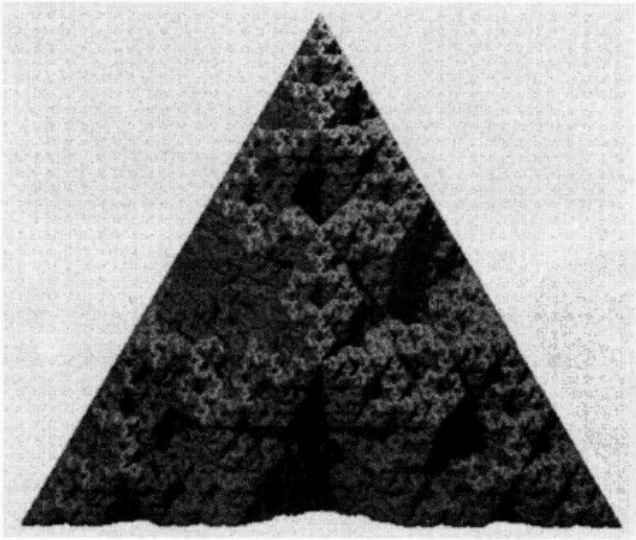


图 5.33　不同参数的三维希尔宾斯基三角体之三

另取 t=–0.02，–0.03，–0.05，生成图像见图 5.34。

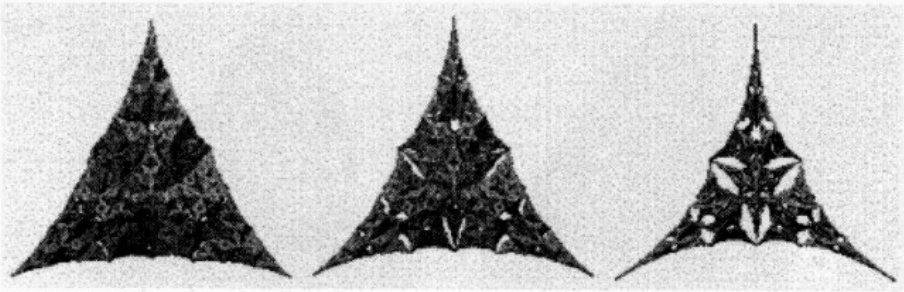
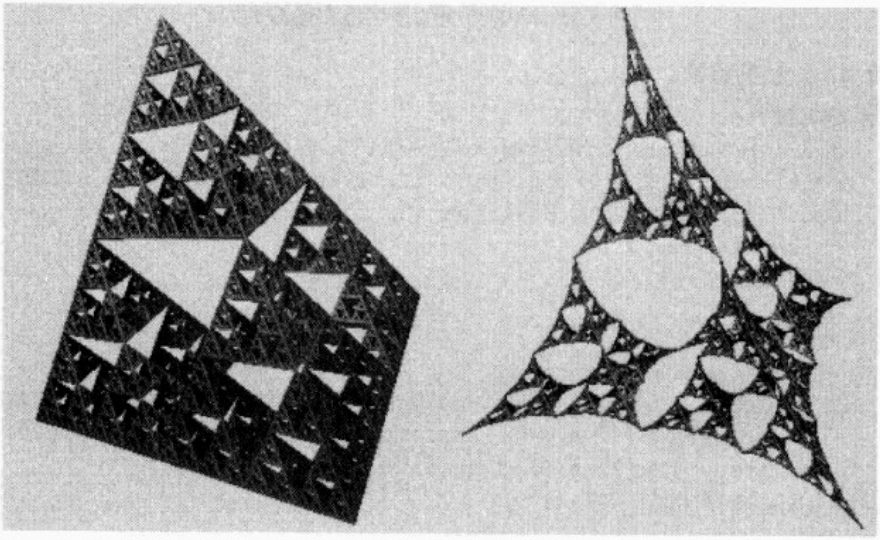


图 5.34　不同参数的三维希尔宾斯基三角体之四

分别改变数据文件的 camera position（0,0, −80）为 camera position（0, −80, −80）和（0, −60, −80），且令 $t=-0.06$，就得到真正立体的希尔宾斯基三角体，见图 5.35（a）和立体三角铁架，见图 5.35（b）。



（a）　　（b）

图 5.35　三维希尔宾斯基三角体之三（a）和三角铁架（b）

再改变数据文件的 camera position（0, 0, –80）为 camera position（0, –60, –80），并令

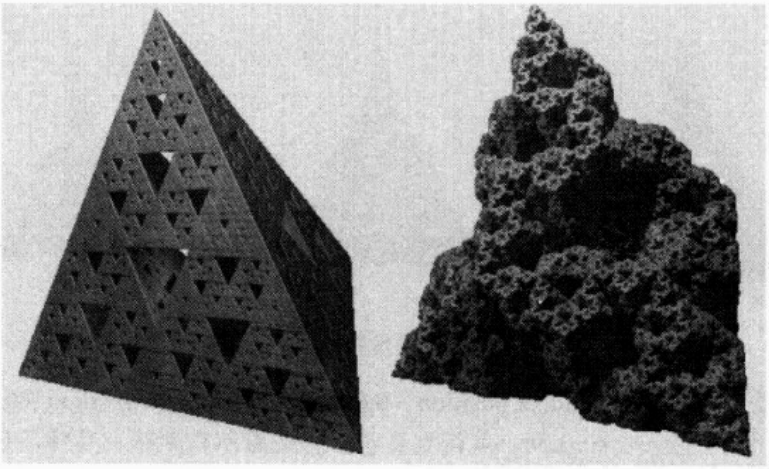
```
t=0;//parameter

k1=3/5-t;
k2=3/5+(4*t);
f=0;
```

和

```
f=180;
```

则得到一座金字塔，见图 5.36（a）和一个怪石头图 5.36（b）。



(a)　　(b)

图 5.36　内有灯光的金字塔和怪石头

例 5.13　十字勋章。

数据文件如下：

```
camera position (0,0,-80) vertical(0,1,0)
light  color (0,1,0) position (1, 1,-3)
light  color (1,1,0) position (3,-1,-1)
ambient(1,1,1,.05)
background(1,1,0,1)

k=1/3;
s=(1+k)/2;
build Cross = (scale(k) + scale((1-k)/2)*(translate( s,0,0)
     +translate(0, s,0)+translate(0,0, s)+translate(-s,0,0)
     +translate(0,-s,0)+translate(0,0,-s)))Cross;
```

文中的 scale 就是压缩因子。

生成的图像见图 5.37。

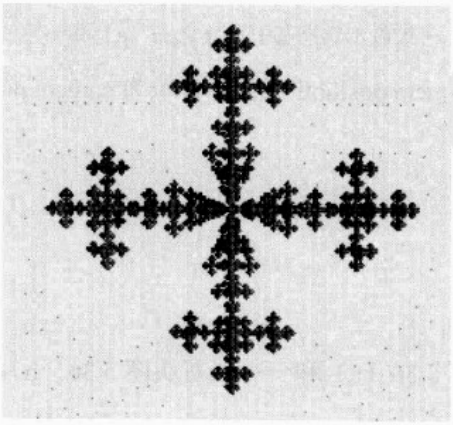


图 5.37　十字勋章图

分别取 k=1/5，1/10 和 1/200，得到图 5.38。

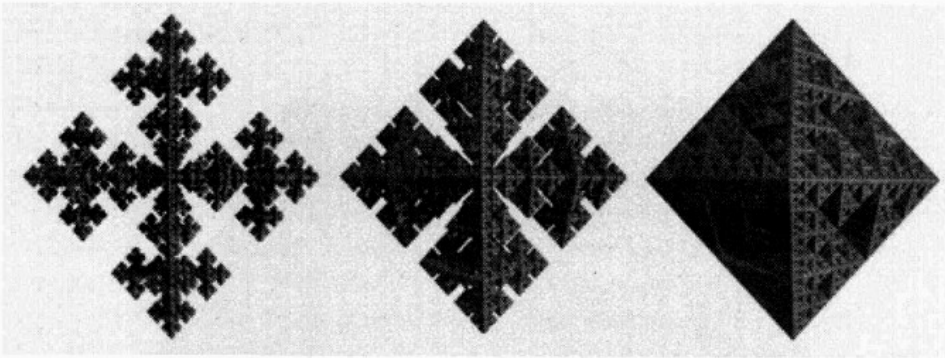


图 5.38　不同参数的十字勋章图

例 5.14　3D 倒 V 字集。

取数据文件为

```
camera position (0,-200,-80) vertical(0,1,0)
light color (0,1,0)   position (1,1,-3)
light color (1,1,0)   position (3,-1,-1)
ambient(1,1,1,.05)
background(0,1,1,1)
// four contractive maps
f1 := translate(-1,-1,0)*scale(1/2)*translate( 1, 1,0);
f2 := translate( 1,-1,0)*scale(1/2)*translate(-1, 1,0);
f3 := translate( 1, 1,0)*scale(1/2)*translate(-1,-1,0);
f4 := translate(-1, 1,0)*scale(1/2)*translate( 1,-1,0);

// four rotation maps
f5 := rotate(0,0,-1,0);
f6 := rotate(0,0,-1,-90);
f7 := rotate(0,0,-1,-180);
f8 := rotate(0,0,-1,-270);

// two maps for the segment in the 2nd set
f9 := translate(-1/2,-1/2,-1/2)*scale(1/2)*translate(1/2,1/2,1/2);
f10:= translate(0,0,-1)*scale(1/2)*translate(0,0,1);

set s1 = (f9+f10)s1;
build s2 = (f1+f2+f3+f4)s2+(f5+f6+f7+f8)s1;
```

这里的旋转是 rotate（x,y,z,angle）。

生成的图像见图 5.39。

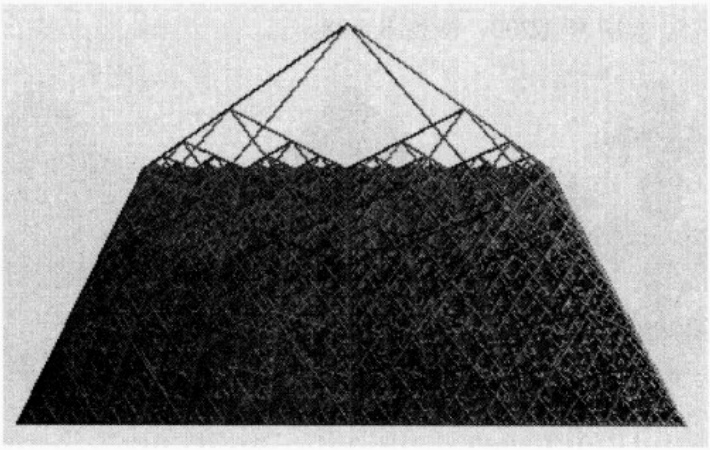


图 5.39　3D 倒 V 字型集之一

另一角度的倒 V 型集见图 5.40。

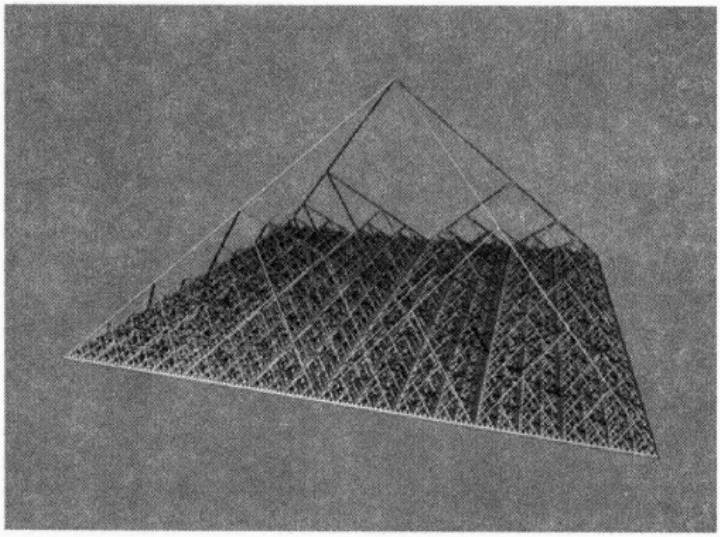


图 5.40　3D 倒 V 字型集之二

例 5.15　分形树。

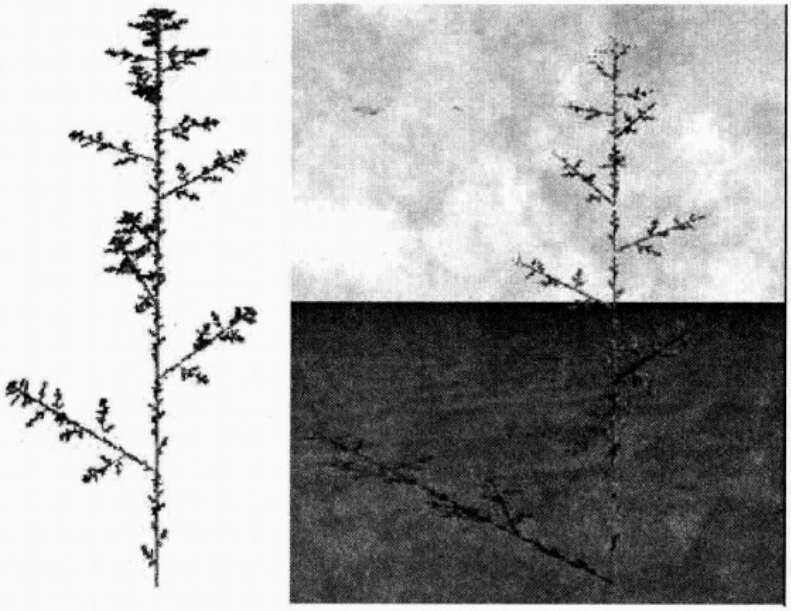
数据文件如下：

```
camera position (0,0,-80) vertical(0,1,0)
light   color (0,1,0) position (1, 1, -3)
light   color (1,1,0) position (3, -1, -1)
ambient(1,1,1,.05)
background(1,1,1,1)

a=.2;
b=.1;
```

```
c=.9;
d=.3;
f0 := scale(b,a,b);
f1 := translate(0,1,0)*rotate(0,1,0,144)*scale(c,1-a,c)
      *translate(0,-1,0);
f2 := translate(0,a,0)*scale(d,d,d)*rotate(0,0,1,60);
build Tree=(f0+f1+f2)Tree;
```

生成图是经加工过的图像，见图 5.41（a），（b）。



（a）　　(b)

图 5.41　IFS 分形树之一

调整数据文件的各个参数 a，b，c，d，得到图 5.42。



图 5.42　IFS 分形树之二

图 5.43 的左面是二维垫片，右面是其三维 IFS 图像。

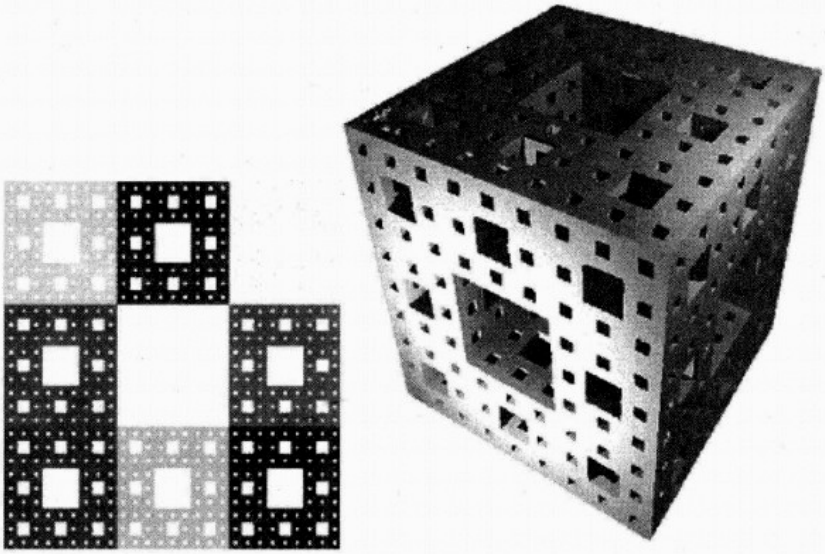


图 5.43　垫片及其立体形

第 6 章　四维空间的四元数分形

6.1　四元数基本理论

我们都熟悉平面上复数 z，其中 $z=a+b\,i$，而 $i=\sqrt{-1}$。无疑，18 世纪以前创立的复数是数学史上的一件大事，那么是否有高维的复数呢？

1843 年，爱尔兰数学家 William R. Hamilton（哈密尔顿）发明了四元数（quaternion）。一个四元数 q 的定义是这样的：

$$q=w+xi+yj+zk \tag{6.1}$$

其中 w,x,y,z 是实数，i,j,k 是虚数，且有

$$i^2=j^2=k^2=ijk=-1 \tag{6.2}$$

q 的模

$$|q|=\sqrt{w^2+x^2+y^2+z^2} \tag{6.3}$$

一个四元数 Q 可以由平面上的两个复数 u,v 来表示:

$$Q=\begin{bmatrix} u & v \\ -\overline{v} & \overline{u} \end{bmatrix}=\begin{bmatrix} a+bi & c+di \\ -c+di & a-bi \end{bmatrix} \tag{6.4}$$

其中 a,b,c,d 是实数。$\overline{u}$，$\overline{v}$ 分别是复数 u,v 的共轭。

一个四元数也能用

$$q=[w,x,y,z] \tag{6.5}$$

来表示。

一个平面上的复数由实部和虚部构成：$z=a\cdot 1+bi$，一个四元数 Q 同样也能由若干部分线性组合而成：

$$Q=wU+xI+yJ+zK \tag{6.6}$$

其中

$$U\equiv\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.7}$$

$$I\equiv\begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix} \tag{6.8}$$

$$J\equiv\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{6.9}$$

$$K\equiv\begin{bmatrix} 0 & i \\ i & 0 \end{bmatrix} \tag{6.10}$$

于是

$$I^2=J^2=K^2=-U \tag{6.11}$$

也就是说，I,J,K 是矩阵方程 $X^2=-U$ 的解，是负单位矩阵的平方根。

一个四元数整系数基的线性组合也叫 Hamilton 整数。在 R^4 空间，四元数的基是如下四个：

$$1=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix} \tag{6.12}$$

$$i=\begin{bmatrix}0&1&0&0\\-1&0&0&0\\0&0&0&1\\0&0&-1&0\end{bmatrix} \tag{6.13}$$

$$j=\begin{bmatrix}0&0&0&-1\\0&0&-1&0\\0&1&0&0\\1&0&0&0\end{bmatrix} \tag{6.14}$$

$$k=\begin{bmatrix}0&0&-1&0\\0&0&0&1\\1&0&0&0\\0&-1&0&0\end{bmatrix} \tag{6.15}$$

四元数的三个虚数之间的运算并不遵从乘法交换律，其运算规则如下：

$$ij=-ji=k \tag{6.16}$$

$$jk=-kj=i \tag{6.17}$$

$$ki=-ik=j \tag{6.18}$$

看起来很像三维空间直角坐标系中三个单位向量 i,j,k 的叉乘关系。

设 $q=w+xi+yj+zk$，则其四元共轭为

$$\overline{q}=w-xi-yj-zk$$

其加法遵从一般规律：

$$q_1+q_2=(w_1+w_2)+(x_1+x_2)i+(y_1+y_2)j+(z_1+z_2)k$$

设 $p=[p_1,p_2,p_3,p_4]$，$q=[q_1,q_2,q_3,q_4]$，其乘法服从

$$\begin{aligned}pq=&(p_1q_1-p_2q_2-p_3q_3-p_4q_4)\\&+(p_1q_2+p_2q_1+p_3q_4-p_4q_3)i\\&+(p_1q_3-p_2q_4+p_3q_1+p_4q_2)j\\&+(p_1q_4+p_2q_3-p_3q_2+p_4q_1)k\end{aligned}$$

q 的模仍然遵从一般复数关系

$$\text{norm}(q)=\sqrt{q\overline{q}}=\sqrt{\overline{q}q} \tag{6.19}$$

且等于公式（6.3）。

一个四元数可以写成一个数量加上一个向量

$$q=(w,\ \boldsymbol{v})$$

其中向量 $\boldsymbol{v}=[x,y,z]$。如此一来，两个四元数的乘法就变得较为简单：

$$\begin{aligned}q_1q_2&=(w_1,\boldsymbol{v}_1)\cdot(w_2,\boldsymbol{v}_2)\\&=(w_1w_2-\boldsymbol{v}_1\cdot\boldsymbol{v}_2,\ \ w_1\boldsymbol{v}_2+w_2\boldsymbol{v}_1+\boldsymbol{v}_1\times\boldsymbol{v}_2)\end{aligned}$$

四元数的除法也遵从复数关系

$$p^{-1}=\frac{\overline{p}}{p\overline{p}}=\frac{p_1-p_2i-p_3j-p_4k}{p_1^2+p_2^2+p_3^2+p_4^2}$$

和

$$\frac{p}{q}=\frac{p\overline{q}}{q\overline{q}}$$

从几何上来讲，四元数代表着时间加三维空间。如果固定实数 w 为常数，则这个四元数就是三维空间的一个变量。

在三维空间画图，旋转是经常要遇到的，而四元数的旋转则较为简单。

设 p,q 是两个非零四元数，则 qpq^{-1} 是三维空间中对 p 的一个旋转。

实际上，因为 $|pq|=|p||q|$，所以 $|p^{-1}|=|\overline{p}|/|p\overline{p}|=|p|/|p|^2=|p|^{-1}$，又 $|qpq^{-1}|=|p|$，这表明 qpq^{-1} 是四维空间的一个正交变换。当 w 是常数时，$qwq^{-1}=w$，故 qpq^{-1} 是三维空间对 p 的一个正交变换。由于 qpq^{-1} 把零点变换到零点，把非零的四元数 p 变换到另一个等长的非零四元数（向量），所以 qpq^{-1} 就是对 p 的一个旋转变换。

最后，我们回忆一下四元数的历史。

1843 年 10 月 16 日的傍晚，爱尔兰数学家哈密尔顿（Hamilton，1805～1865）和他的妻子沿着皇家运河步行，途中经过布鲁哈姆桥（Brougham Bridge）时，突然放慢了脚步。原来他此时正在思考他久久不能解决的问题：两个来自固定转动轴，一个来自规定的转动角度，第四个来自规定向量的伸缩。但在构造新代数的过程中，由于受传统观念的影响，不肯放弃乘法交换律，故屡受挫折。哈密尔顿盲目地相信，普通代数的最重要规律必定继续存在于他寻找的代数中。能否让交换律不成立呢？比如说，$A\times B$ 不等于 $B\times A$，而是（$-B\times A$）。这个想法太大胆了，他感到异常地兴奋和激动，以至于他马上拿出笔记本，把这一刹那的思想火花记了下来。这思想火花就是 i,j,k 之间的基本关系，即四元数的基本乘法公式。兴奋之下，他取出小刀在桥的一块石头上刻下他的发现：$i^2=j^2=k^2=ijk=-1$。四元数的出现，推倒了传统代数的束缚，故有数学史上里程碑的美誉。

哈密尔顿发现了长期探索的结果，但这毕竟太离奇太神秘了，因为它是四维的。哈密尔顿做的头件事就是摆脱第四维，让它为零并称其是本征四元数（proper quaternion）。到 19 世纪末，四元数还是被看作很新奇的东西。但到 20 世纪初，耶鲁大学的济勃斯（Gibbs）教授发现了四元数的用处，特别是用于球面的旋转上。后来爱因斯坦发现了四维的用处。用四元数去产生三维分形图像的先行者是美国的斯蒂文（R. T. Stevens），不过其最初的四元数茹利亚集画得并不理想。后来澳大利亚的保罗（Boorke Paul，保罗是他自己翻译的中文名字）等人发展了他的工作。

6.2　三维立体绘图的基本原理

目前，在计算机图形学（computer graphics，简称 CG）领域中，研究最活跃的是三维计算机图形学，其最主要的研究方向有

1）立体对象的输入方式；

2）建模方法；

3）造型方法；

4）CG 和 CAD 的应用（视频制作、CAD/CAM 系统等）；

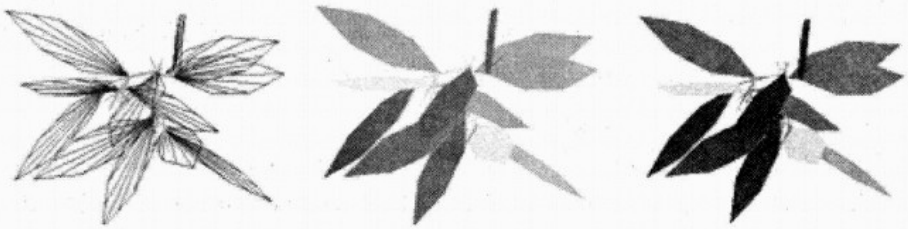
5）CG 的标准化。

在这些方向中，立体对象的输入方式的研究最难，而建模和造型是核心。甚至可以说

三维计算机图形学=建模+造型

换言之，对于输入到计算机的数据进行三维立体建模是根据可视化处理的造型操作来实现的。

建模：主要的模型有线框架结构模型（wire farme model）、实体模型（solid model）和表面模型（surface model）。分别见图 6.1 的（a）、（b）、（c）。



（a）　　（b）　　（c）

图 6.1　三维图像的三种模型

造型：主要方法有

1）面消隐和线消隐法；

2）明暗处理；

3）阴影生成处理；

4）具有真实感的造型（纹理匹配法，光线追踪法和辐射度算法）；

5）建模加造型处理（分形和实体造型）。

由于人们所见的物体只是其表面部分，所以消隐法特别重要。最常用的是排列优先法中的深度缓存（Z-buffer）法。深度缓存法是物体在从三维空间向二维空间投影时进行面消隐的最简单的算法。它利用被称作 Z-buffer 的矩阵对每个像素进行处理，以期达到面消隐的目的。记 $\mathrm{depth}(x,y)$ 表示深度缓存数据，$\mathrm{intensity}(x,y)$ 表示图像的帧缓存数据，则深度缓存法的算法如下：

1）背景的颜色 B 在初始化时设定为

$$\mathrm{depth}(x,y)=\infty$$

$$\mathrm{intensity}(x,y)=B$$

2）场景中的各个多边形向画面投影时，寻找多边形区域中所包含的所有像素 (x,y)；

3）计算每个像素（x,y）的深度 Z；

4）如果 $Z \leqslant \text{depth}(x,y)$，则包含这个像素（$x,y$）的多边形在之前处理过的多边形的内部，表示最接近观察者，可将 Z 代入 $\text{depth}(x,y)$，随之将多边形的灰度代入 $\text{intensity}(x,y)$。如果 $Z>\text{depth}(x,y)$，则表示存在更接近观察者的多边形，所以不做任何操作。

我们所绘的虽然是三维空间中的物体，但是只能在二维平面中表现，我们在一个时刻所看到的只能是空间中物体在观测点所在平面上的投影。物体中有很多点的投影点是相重合的，投影点相重合的点都在与平面法向量平行的直线上（也就是我们的观测方向），我们能看到的是离平面最近的一个点，所以我们只要找到这个点就行了。为此，我们先建立一个空间模型，见图 6.2。

在 $OXYZ$ 空间中，我们先选择一个观测点 P 和一个观测方向 n_0，于是得到投影平面 A。要从观测点的一个正方形邻域中每个点沿法向量方向寻找物体中的第一个点，实际上问题转化为空间坐标系和平面 A 上坐标系的转换。

首先，我们在平面 A 上以 P 为原点建立坐标向量，n,u,v 应满足：

$$n = u \times v$$

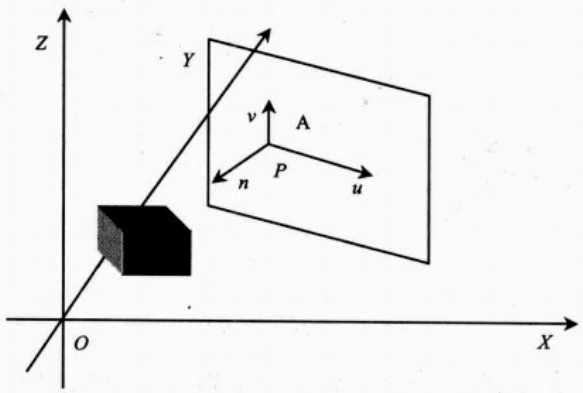


图 6.2　空间模型图示

但 u,v 的方向并不能确定，为了视觉上看起来更习惯，我们选择让 u 平行于 XOY 平面。这样有方程组：

$$\left.\begin{aligned} u \cdot v &= 0 \\ u \cdot n &= 0 \\ |u| &= 1 \end{aligned}\right\}$$

由此可以解得 u,v，另外我们可以得到两个坐标系坐标的转换公式：

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + x \cdot u + y \cdot v$$

其中 (x,y) 是 A 平面的坐标，(x',y',z') 是 $OXYZ$ 中的坐标。

对于 A 平面上一点，先找到它在 $OXYZ$ 中的坐标 (x',y',z')，沿法向量方向逐步搜索，步长可以自定义。用逃逸时间法确定它是否属于集合，得到的第一个属于集合的点就是

我们要求的点。

这样我们就找到了可以看见的物体上的所有的点。为了体现出立体效果，还需要用到光线追踪原理，叙述如下：

对曲面上某一点，我们计算某给定点光源在其上的光照效果，目标是算出该点的亮度值。首先算出该点的法向，因为我们只有曲面上一些离散的点，故可以用差分的方法，见图 6.3。下方的浅色格子表示 XOY 平面，垂直粗线的上端点表示曲面上的点，粗线高度各自不同表明每个点在 Z 方向的“深度”不同。研究中间更粗的粗线表示的点，其法向可由它前后左右的四个点大致决定：把他们用图中黑线连接，就得到一个平面，该平面的法向被认为是该点的法向。

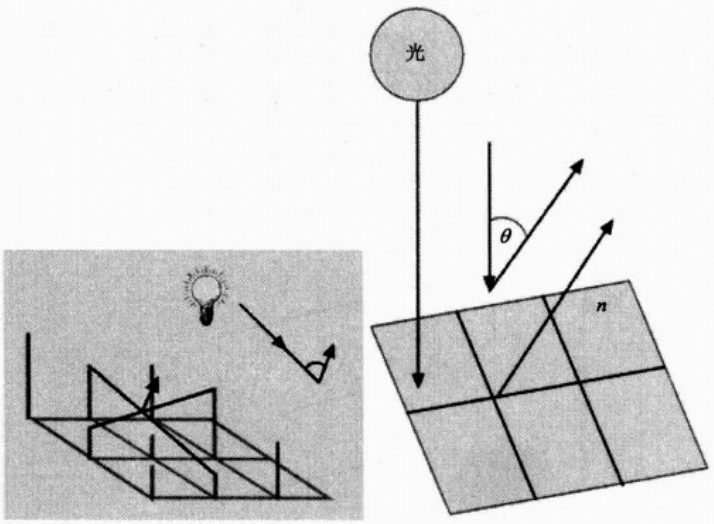


图 6.3　光线追踪原理图

这里的光源是一给定的点，在某一点光照的效果是两个向量决定的：由光源到该点构成的向量，和刚才算出的该点的法向量。计算他们的夹角（在$-\pi$到π之间），并取绝对值。不难发现，夹角为 0 时，目标点背对光源。夹角为π时，目标点正对光源。显然，夹角在 $0-\pi/2$ 之间的点应涂上黑色。从$\pi/2$到π涂的颜色应越来越亮。

6.3　四元数分形

6.3.1　四元数曼德勃罗集

记 q 和 c 为四元数，按第 2 章中曼德勃罗集的迭代法

$$q_{n+1} = q_n^2 + c, \qquad n = 0,1,2,\cdots \tag{6.20}$$

就得到一幅三维四元数（把实数 w 看作常数，并认定其为第四维）曼德勃罗集了，见图 6.4。

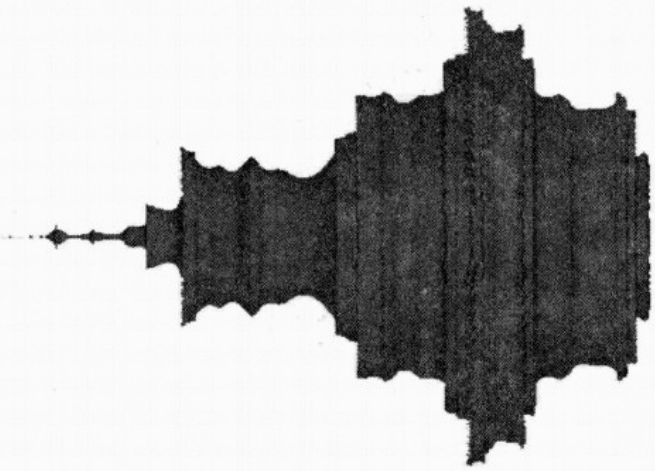
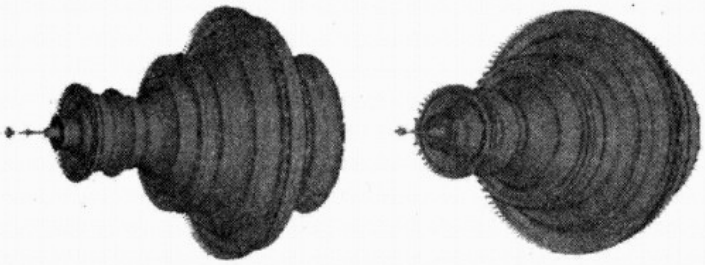


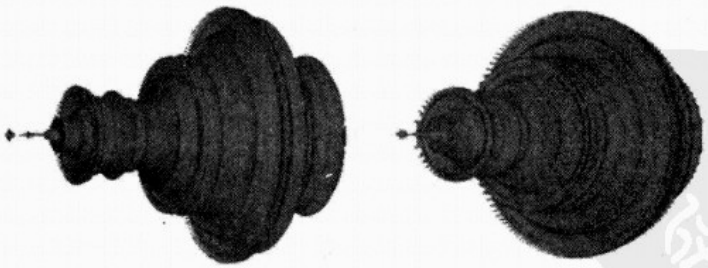
图 6.4　四元数曼德勃罗集之一

平面上的二维曼德勃罗集是唯一的($z_0=(0,0)$)，而三维空间里由于视角不同，当然会有很多不同画面的曼德勃罗集。图 6.5 和图 6.6 是图 6.4 所示的图像绕 Y 轴不同角度的四幅曼德勃罗集。



绕 Y 轴 20°　　绕 Y 轴 40°

图 6.5　四元数曼德勃罗集之二



绕 Y 轴 200°　　绕 Y 轴 240°

图 6.6　四元数曼德勃罗集之三

2D 曼德勃罗集是 X 轴对称的，四元数曼德勃罗集同样轴对称。

2D 曼德勃罗集的初值是 $z_0=(0,0)$，如换成其他初值，图形未必好看。而对于四元数曼德勃罗集，若令 $q_0=(1,0,0,0)$，则有图 6.7。

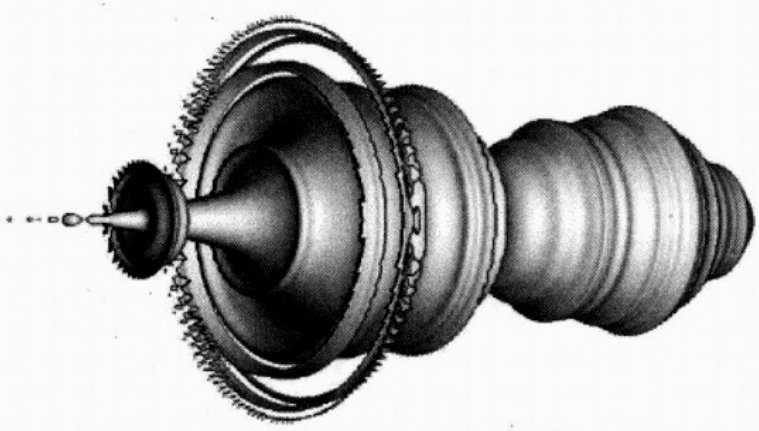


图 6.7　四元数曼德勃罗集之四

四元数曼德勃罗集的画图计算可参看程序 6.1 茹利亚集的计算,只要稍作修改就能画出四元数曼德勃罗集来。

6.3.2　四元数茹利亚集

四元数茹利亚集是 4 维空间中的四元数茹利亚集合，同样使用公式

$$q_{n+1} = q_n^2 + c, \qquad n = 0,1,2,\cdots$$

其中 $q_n(n = 0,1,2,\cdots)$ 和 c 都是四元数。生成四元数 Julia 集的方法与普通复数情况类似，只需迭代公式（6.20），只是 c 是一个固定的四元数而已。迭代时要观察 $|q_n|$ 的敛散情况。

把第四维看作时间，可以认为我们生成的是一个三维动画片。实际操作时，一般取第四维为常数，这样我们得到的是静止的三维图像。程序主体结构与二维情况类似，我们用多重循环扫描一个三维立方体内部所有的点，对他们做 calc_1（见程序 6.1）函数中的运算，就知道该点是否属于茹利亚集了。做法如下：

对每一个 y

　对每一个 x

　　对每一个 z

作迭代

$$q_{n+1} = q_n^2 + c, \qquad n = 0,1,2,\cdots$$

其中 $q_n = [0, x_n, y_n, z_n]$。当 $n = M$ 时，q_n 仍属于集合的话就退出这一层循环。

我们并不计算 z 轴方向上的所有点，那样速度太慢。对于给定的一组（x,y），只要沿 z 轴方向找到第一个属于 Julia 集合的点就够了，其他的点被挡住，让看不见。这样，我们得到一张高度表，程序中用二维数组 z_buffer 存储 *XOY* 平面上每一点的“高度”。在程序中，我们用到了光线追踪（Ray-Tracing）原理，见前面叙述。

这里给出一个完整的四元数茹利亚集合程序：QuaterJ.cpp。可以在 DOS 下用 Turbo C++ 3.0 就可运行。尽管在很多语句后面有对该语句的解释，程序中间，我们还是做了一些必要的说明。当然这些汉文字不属于程序的任何部分。

程序 6.1　四元数茹利亚集。

```
#include <math.h>
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
  int zant=450;   //z-resolution. bigger zant -> better resolution
  int zant1=25;   //z-resolution. bigger zant -> better resolution
  int pixsize=2, vissize=2;  //bigger value -> worse resolution, 1=<
   //& <=4
  //choose pixsize=visize to make full screen picture
```

zant 和 zant1 控制 z 方向上划分的深度层数。先以较大的步长向前搜索 z 方向，直到“走过头”以后，再用较小的步长往回走，确定确切位置。注意这里 zant 和 zant1 不是步长大小，而是划分的层数，故 zant 越大效果越好。pixsize 和 vissize 控制图像精度。比如，pixsize 设为 4，则每四个点才计算一次，这样可得到粗糙的预览图像。vissize 表示每个点的大小。这两个值越小图像越清晰。注意只有二者相等时图像才恰好充满屏幕。

```
double xmin=-1.66, xmax=1.6;
double ymin=-1, ymax=1;
double zmin=-2.1, zmax=1.7; //take care here, or you'll get some black area
int iter=10;  //for julia set better not exceed 15
double lightx=-5, lighty=5, lightz=-5;
double vx=0, vy=0, vz=0;  //rotation angles
```

这几个参数很好理解，设置观察区域（是一个立方体）和迭代次数 iter，一般不用取太大就可得到很好的图像。light 是光源位置，vx,vy,vz 是空间的旋转角度。

```
double cr=-0.83;  //constant real value of c
double ci=0.3;  //constant imaginary(1) value of c
double cj=0.3;  //constant imaginary(2) value of c
double ck=0.05;   //constant imaginary(3) value of c
double wk=0;   //4th dimension
```

对于 $q=q^2+c$，这里设置四元数 c 的各个分量值和四元数 q 的第四维常数 wk。只需更改上述四个参数，就可得到多姿多彩的三维图像。

```
int background = 0;
int bgcolor_r=255, bgcolor_g=255, bgcolor_b=255;
int maxcolor = 16;
int color;
```

这里设置背景色 bgcolor，background 是指是否对背景做光照效果处理。

```
int sx,sy;
double dx,dy,dz;
double origx, origy, origz;
double rminx, rminy, rminz;
double dxx, dxy, dxz;
double dyx, dyy, dyz;
double dzx, dzy, dzz;
double dzx1, dzy1, dzz1;
double tempx, tempy, tempz;
double cosx,cosy,cosz,sinx,siny,sinz;
```

```
double *z_buffer[640];
int buffer_y;
```

这些是程序中用到的中间变量，值得指出的是 z_buffer 用于存放 Z 轴信息。

```
void rotate3D(double &x,double &y,double &z)
{
  x-=origx;y-=origy;z-=origz;
  double xy=y*cosx-z*sinx;
  double xz=y*sinx+z*cosx;
  double xx=x;
  x=xx;
  y=xy;
  z=xz;
  double yx=x*cosy+z*siny;
  double yz=-x*siny+z*cosy;
  x=yx;
  z=yz;
  double zx=x*cosz-y*sinz;
  double zy=x*sinz+y*cosz;
  x=zx;
  y=zy;
  x+=origx;y+=origy;z+=origz;
}
```

这个函数以点 org 为中心，以 vx，vy，vz 为旋转角度，旋转空间中一点。

```
void rotatevalues()
{
  rminx=xmin;rminy=ymin;rminz=zmin;
  rotate3D(rminx, rminy, rminz);
  tempx=xmax;tempy=ymin;tempz=zmin;
  rotate3D(tempx, tempy, tempz);
  dxx=(tempx-rminx)/sx;dxy=(tempy-rminy)/sx;dxz=(tempz-rminz)/sx;
  tempx=xmin;tempy=ymax;tempz=zmin;
  rotate3D(tempx, tempy, tempz);
  dyx=(tempx-rminx)/sy;dyy=(tempy-rminy)/sy;dyz=(tempz-rminz)/sy;
  tempx=xmin;tempy=ymin;tempz=zmax;
  rotate3D(tempx, tempy, tempz);

 dzx=(tempx-rminx)/zant;dzy=(tempy-rminy)/zant;dzz=(tempz-rminz)/
 zant;
  dzx1=dzx/zant1;dzy1=dzy/zant1;dzz1=dzz/zant1;
}
```

这个函数以 rotate3D 为基础，用于旋转整个空间，并初始化单位向量。三维空间的旋转变换属于计算机图形学的内容，因篇幅所限不再解释，直接使用即可。

```
double calc_l(double x, double y, double z)
{       //  (x,y,z,w)^2 =
    //              ( x*x - y*y - z*z - w*w ,
    //                x*y + y*x + z*w - w*z ,
    //                x*z + z*x - y*w + w*y ,
    //                x*w + w*x + y*z - z*y ) }
  double lengde;
```

```
 double temp;
 double w=wk;
 int m=0;
 do {
  temp=x+x;
  x=x*x-y*y-z*z-w*w+cr;
  y=temp*y+ci;
  z=temp*z+cj;
  w=temp*w+ck;
  m++;
  lengde=x*x+y*y+z*z+w*w;
 } while ((m<iter) && (lengde<4));
return lengde;
}
```

这是主要的计算过程。函数参数为一四元数(x,y,z,w)，经过循环内部迭代过程返回其模的平方。m 表示迭代次数，如果在最多 iter 次迭代过程中 q 的模大于 2，即模的平方大于 4，则认为不收敛，退出循环. 返回值就是 q 的模的平方，可根据它是否大于 4 判断敛散性。

```
double calc_angle(double w,double e,double n,double s,double
   cx,double cy,double cz)
{
  double lightdx=cx-lightx;
  double lightdy=cy-lighty;
  double lightdz=cz-lightz;
  double
   lightlength=sqrt(lightdx*lightdx+lightdy*lightdy+lightdz*lightdz);
  double fx=/*(0)*(s-n)*/-(e-w)*(dy+dy);
  double fy=/*(e-w)*(0)*/-(dx+dx)*(s-n);
  double fz=(dx+dx)*(dy+dy)/*-(0)*(0)*/;
  double flength=sqrt(fx*fx+fy*fy+fz*fz);
  double c=(fx*lightdx+fy*lightdy+fz*lightdz)/(flength*lightlength);
  return c;
}
```

这段程序计算光源到目标点的向量和目标点法向量，最终返回二者夹角. 这里主要用到了高等几何的一些知识，不再解释。

```
void update_buffer(int y)
{
  double a;
  int t;

  for (t=1; t<sx; t++) {
    for (int i=1; i<=8; i++) {
    if ((y+i)<sy) {
  a=calc_angle(z_buffer[t-1][i],z_buffer[t+1][i],z_buffer[t][i-1],
  z_buffer[t][i+1],t*dx+xmin,(y+i)*dy+ymin,z_buffer[t][i]);
        if ((z_buffer[t][i]>zmax) && (background==0)) {
          setfillstyle(1,0);
        } else if (a<0) {
```

```
            setfillstyle(1,1);
          } else {
            setfillstyle(1,1+(maxcolor-1)*a);
```

根据计算的夹角 a 对当前点选择填充色，以达到光照效果。

```
        }

 bar(t*vissize,(y+i)*vissize,t*vissize+vissize-1,(y+i)*vissize+vissiz
 e-1);
```

用 bar 函数绘图而不是 putpixel 的原因是，用 bar 可以通过控制 vissize 画出较大的“点”。

```
          }
         }
       }
       for (t=0; t<640; t++) {
        z_buffer[t][0]=z_buffer[t][8];
        z_buffer[t][1]=z_buffer[t][9];
       }
       buffer_y=2;
     }
```

这个函数进行渲染过程，重新绘制了所有点。考虑到在 Dos 下编写程序时，数组使用的内存大小超过 64k 的话必须用特殊方法处理（far 指针），较为麻烦，我们 z_buffer 的大小仅为 640*10。每计算完毕 10 行点的 q 值以后，在本函数中完成渲染工作并显示出来，然后再把第 8 行点复制到第 0 行，把第 9 行点复制到第 1 行，以便下次使用。

```
    void main()
    {
     int pz, pz1;
     double l;
     int gdriver = VGA, gmode=VGAHI, errorcode;
      errorcode = 1;
      if (errorcode < 0) {
       printf("Graphics error: %s\n", grapherrormsg(errorcode));
       exit(1);
      }

      initgraph(&gdriver, &gmode, "");
      errorcode = graphresult();

      if (errorcode != grOk) {
       printf("Graphics error: %s\n", grapherrormsg(errorcode));
       exit(1);
      }
```

以上是图形界面的初始化工作。

```
    for (int i=0; i<640; i++)
        z_buffer[i]=new double[10];
```

分配 z_buffer 使用的内存。

```
    for (i=0; i<16; i++) {
```

```
    setrgbpalette(i, 0, i*4, 0);
    setpalette(i, i);
  }
  setrgbpalette(0, bgcolor_r, bgcolor_g, bgcolor_b);
  setpalette(0, 0);
```

设置调色板为渐变的绿色，调色板中第一种颜色为背景色。

```
  sx=getmaxx()/pixsize;sy=getmaxy()/pixsize;
  dx=(xmax-xmin)/sx;
  dy=(ymax-ymin)/sy;
  dz=(zmax-zmin)/zant;
  double dz1=dz/zant1;
  origx=(xmin+xmax)/2;
  origy=(ymin+ymax)/2;
  origz=(zmin+zmax)/2;

  vx=vx/180*3.14159265;
  vy=vy/180*3.14159265;
  vz=vz/180*3.14159265;

  cosx=cos(vx);cosy=cos(vy);cosz=cos(vz);
  sinx=sin(vx);siny=sin(vy);sinz=sin(vz);

  rotatevalues();
```

计算屏幕宽度、步长、原点等基本常数，最后根据 *vx,vy,vz* 旋转三维空间，同时计算出单位向量。

```
  buffer_y=0;
  for (int py=0; py<=sy; py++) {
     for (int px=0; px<=sx; px++) {
      tempx=rminx+px*dxx+py*dyx/*+pz*dzx*/;
      tempy=rminy+px*dxy+py*dyy/*+pz*dzy*/;
      tempz=rminz+px*dxz+py*dyz/*+pz*dzz*/;
      pz=0;
      do {
        tempx+=dzx;
        tempy+=dzy;
        tempz+=dzz;
        l=calc_l(tempx,tempy,tempz);
        pz++;
      } while ((l>2) && (pz<zant));
              //向前扫描结束
      pz1=0;
      do {
        pz1++;
        tempx-=dzx1;
        tempy-=dzy1;
        tempz-=dzz1;
        l=calc_l(tempx,tempy,tempz);
      } while (l<2);
```

```
        //向后扫描结束

        if (pz < zant)
          z_buffer[px][buffer_y]=zmin+pz*dz-pz1*dz1;
        else
          z_buffer[px][buffer_y]=zmax+dz;
        //计算出了 pz 以后，设置 z_buffer
       color=15-pz/10;
       if (color%16==0)
         color=1;
       setfillstyle(1,color);

        bar(px*vissize,py*vissize,px*vissize+vissize-1,py*vissize+vissize
      -1);
      if (kbhit()) break;
       }
      buffer_y++;
      if (buffer_y==10) update_buffer(py-9);
      if (kbhit()) break;
   }
     if (!kbhit()) {
     update_buffer(py-buffer_y);
       }
       getch();
       closegraph();

     for (i=0; i<640; i++)
        delete []z_buffer[i];
}
```

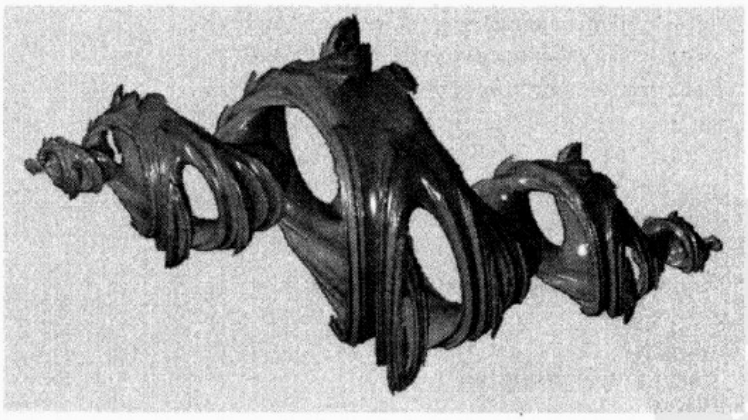


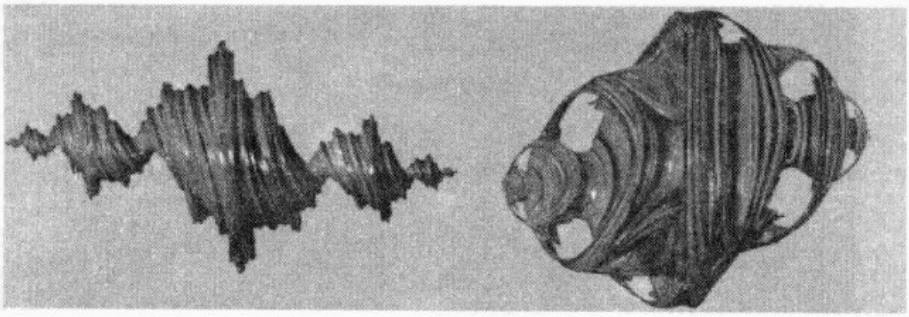
图 6.8　程序 6.1 生成的四元数茹利亚集

以上是程序主体。要点在于处理 Z 轴上的点时，用不同的步长扫描了两次，然后才确定 z_buffer 的值。每计算 10 次，即 buffer_y＝10 时重设 z_buffer，并重置 buffer_y 的值。

程序的最后是关闭图形方式并释放内存。

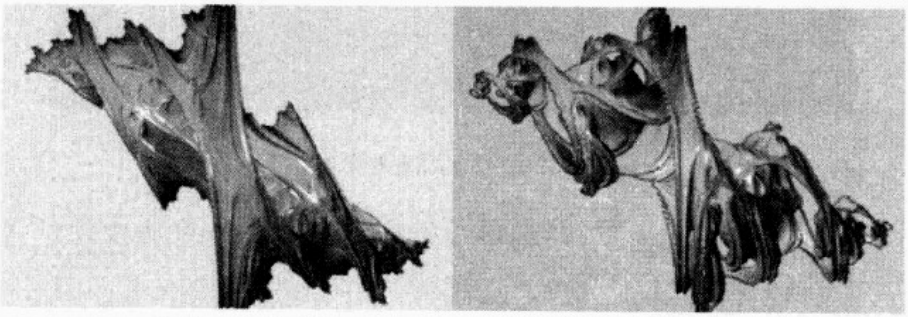
该程序产生的图形见图 6.8。

图 6.9、图 6.10 和图 6.11 是不同常（四元数）数 c（图下方有数字表示）所对应的图像。



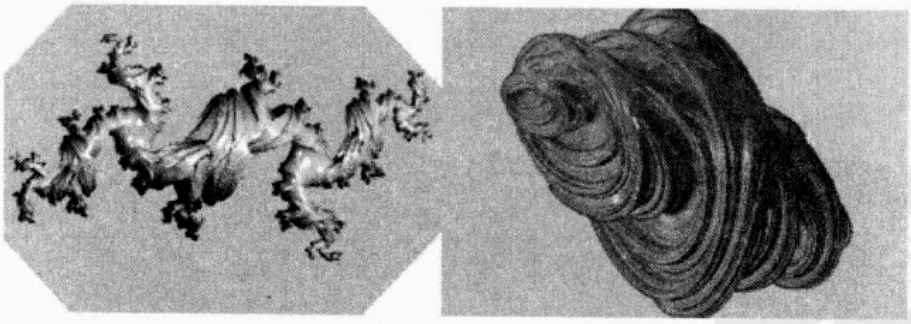
（−1，0.2，0，0）　　（−0.125，−0.256，0.847，0.0895）

图 6.9　四元数茹利亚集之一



（−0.2，0.8，0，0）　　（−0.4564，0.6705，0.3224，−0.0924）

图 6.10　四元数茹利亚集之二



（−0.7，0.45，0.77，0.05）　　（0，−0.8794，0.5924，0.7）

图 6.11　四元数茹利亚集之三

四元数茹利亚集也是无穷无尽的，且由于在三维空间，其变化就更千变万化。下面选择几幅图像，有单独存在的，也有贴在其他画面上的，见彩色图片 6 及图 6.12～图 6.14。

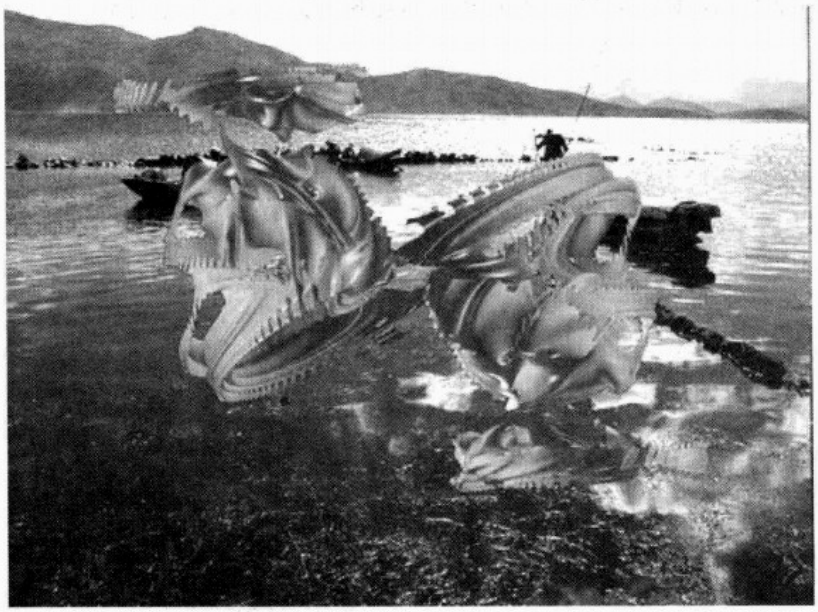


图 6.12　四元数茹利亚集之四

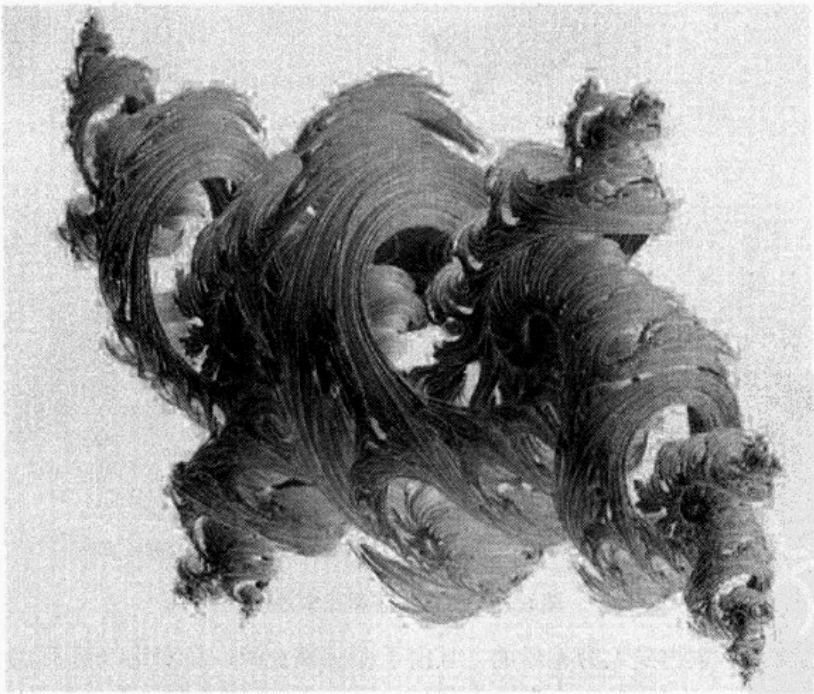


图 6.13　四元数茹利亚集之五

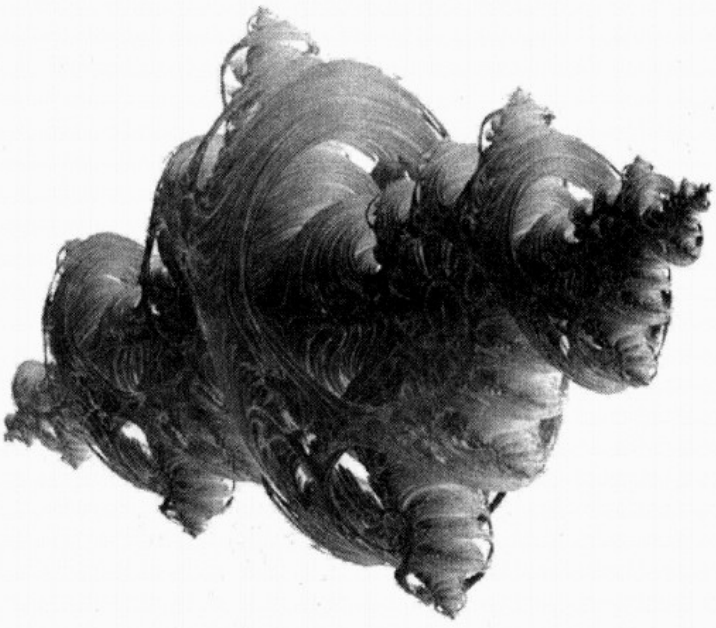


图 6.14　四元数茹利亚集之六

6.3.3　四元数多项式和有理式的广义茹利亚分形

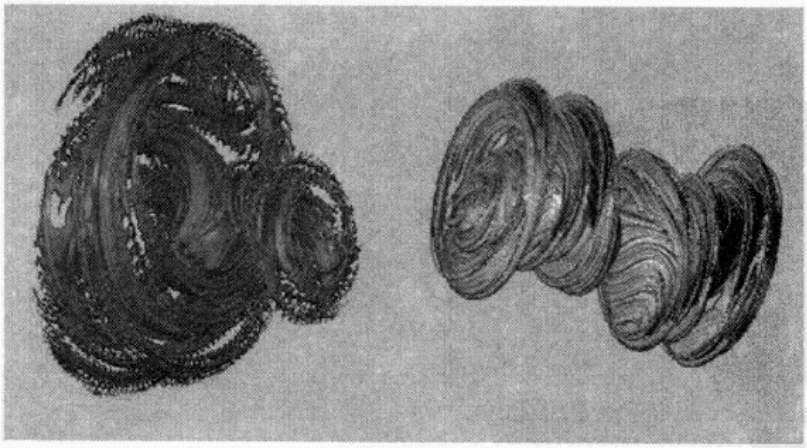
由于以四元数为自变量的函数计算过于困难，目前我们还仅仅处理 q 的多项式和有理式。如果说茹利亚分形来自函数 $f(q)=q^2+c$，则当

$$f(q)=q^3+c,\quad c=(-0.6564,0.5106,0.0224,0.0176)$$

时，有图 6.15（a）。如果

$$f(q)=q^4+q^2+c,\quad c=(0.7264,-0.4206,-0.0540,-0.0824)$$

则有图 6.15（b）。



（a）　　（b）

图 6.15　多项式广义茹利亚分形之一

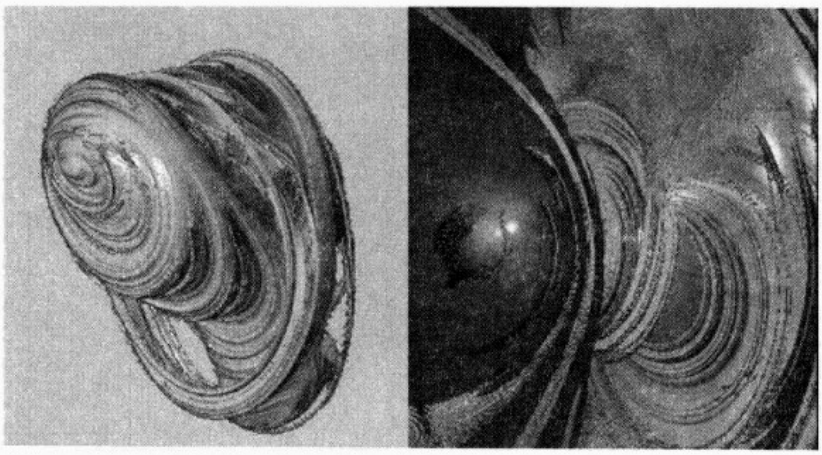
若令

$$f(q)=(q^2-c^2)^2+cq\ ,\quad c=(0.4264,-0.4706,-0.0740,-0.1824)$$

可生成图 6.16（a）。再令有理分式

$$f(q)=q(q^2-q)/(q^2-1)-c\ ,\quad c=(0.5364,-0.4604,-0.4976,0.4976)$$

有图 6.16（b）。



（a）　　（b）

图 6.16　多项式和有理式广义茹利亚分形之二

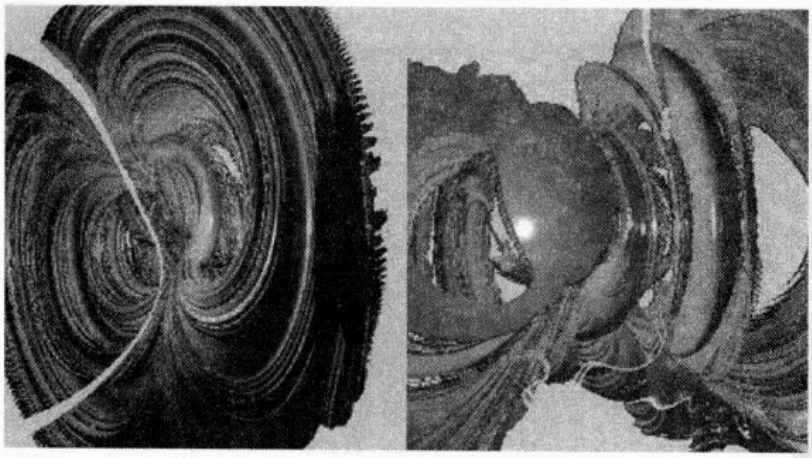
再设有理分式

$$f(q)=q(q^2-c)/(q-1)+c$$

当常数 c 分别为

$$c=(-0.1664,0.2206,-0.4976,-0.3024)$$
$$c=(-0.3464,0.3606,-0.4976,-0.0576)$$

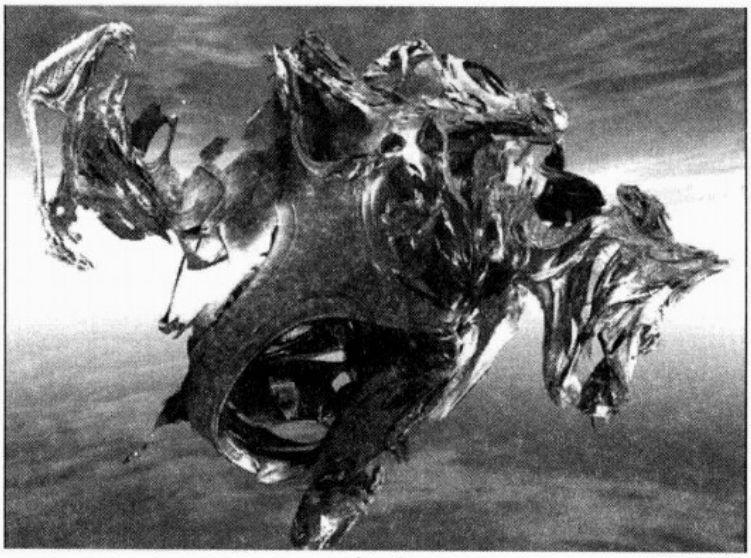
时，有图 6.17。



（a）　　（b）

图 6.17　有理式广义茹利亚分形之三

彩色图片 7（左）和图 6.18～图 6.20 是对四元数多项式函数的迭代结果（经过加色处理）。



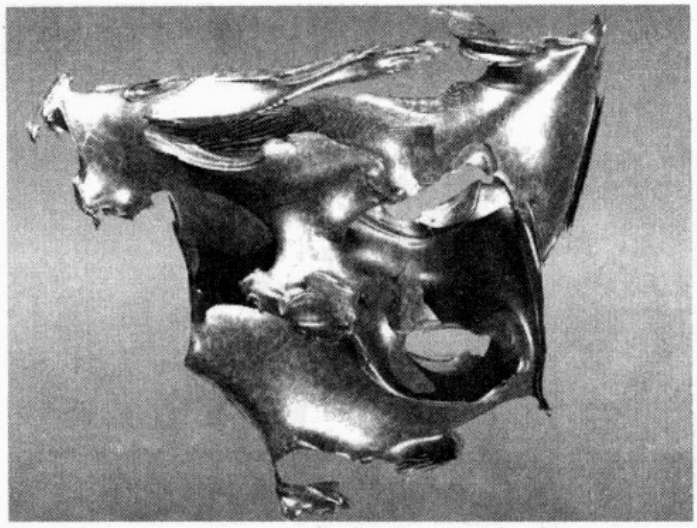
$q^2-q^3+c_1$

图 6.18　四元数多项式分形之一



$q^2-q^3-c_2$

图 6.19　四元数多项式分形之二



$q^2-q^3-c_3$

图 6.20　四元数多项式分形之三

6.3.4　四元数牛顿分形

令 $f(q)$ 是以四元数 q 为自变量的多项式，按牛顿迭代法，我们有迭代公式

$$q_{n+1}=q_n-\frac{f(q_n)}{f'(q_n)},\quad n=0,1,2,3,\cdots \tag{6.21}$$

绘制四元数牛顿分形的程序可以从程序 6.1 演化过来，即认定其第四维 $w=0$，从而进行三维空间的运算，但有几个重要问题必须处理好。

1）如何判断找到了方程的根？

固定一个的很小的正数 ε，在用牛顿迭代公式进行求根运算时，当两次迭代结果的差的模 $|q_{n+1}-q_n|$ 的平方不大于这个正数时，就认为已经找到了方程的根，并立即停止以当前 (x,y,z) 为初始条件的迭代运算，继续计算下一个点，否则继续进行迭代运算。如果超过了最大的迭代次数，迭代前后的结果差别很大，则认为以当前 (x,y,z) 为初始条件无法找到方程的根。

2）如何判断方程的根是否突变？

设定一个阈值（逃逸条件），以临近的两个点为初始条件进行迭代运算求出方程的根，计算这两个根的差的模的平方，判断结果是否大于设定的阈值。如果大于阈值，则认为根发生了突变，这一点需要在屏幕上绘出，反之继续搜索。

3）鉴于牛顿集的运算量要明显多于曼德勃罗集和茹利亚集，所以我们只好牺牲一些 z 轴精度来减少运算时间，对 z 轴的遍历不再进行两种步长的计算，只对 z 轴遍历 150 点。

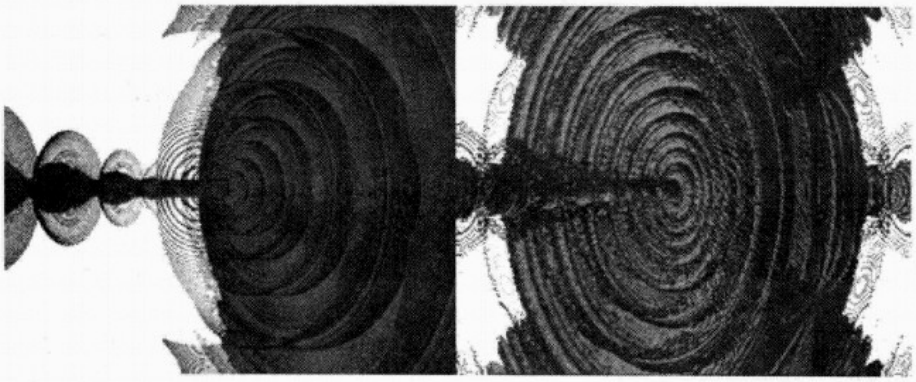
算法：

```
for x=0 to n
for y=0 to n
//搜索 z:
for z=0 to n
//以当前(x,y,z)作为初始点计算 newton 方程的根，设根为 r。
```

```
//然后随着 z 的增加，比较这次算出来的 r 与上次算的 r0，是否是同一个根。
//如果不同，就记录该点，退出循环。
r0 = r;
r = solve_newton(x,y,z);
if (different_root(r, r0)) {
  z_buffer[x][y]=z;
  break;
}
```

下面分别对不同多项式绘制其牛顿分形。

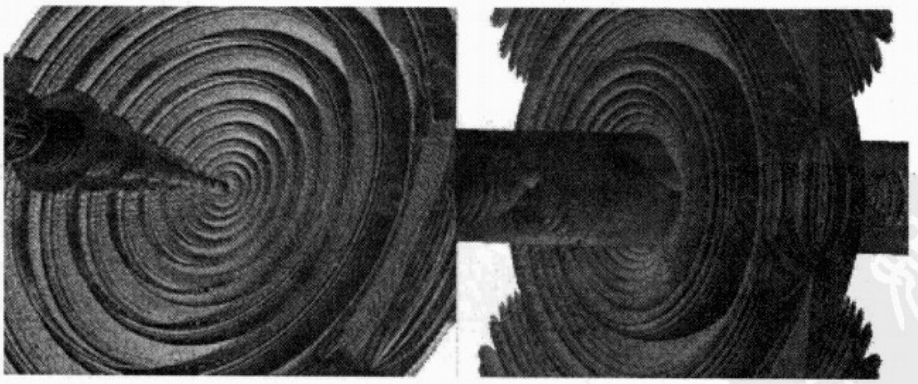
1）设 $f(q)=q^3-c$，且令 $c_1=(2,0,0,0), c_2=(-10,0,0,0)$，画得其牛顿分形分别见图 6.21 的(a)和(b)。



(a)　　(b)

图 6.21　三次多项式牛顿分形

2）设 $f(q)=q^4-c$，令 $c_1=(-2,0,0,0), c_2=(-10000,0,0,0)$，得其图像，见图 6.22（a）和(b)。



(a)　　(b)

图 6.22　四次多项式牛顿分形之一

令 $c=(2,0,0,0)$，得到图 6.23。

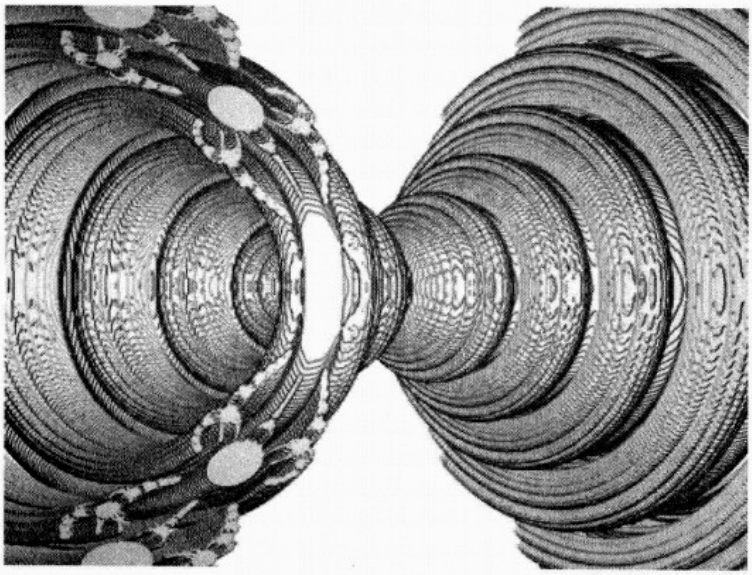


图 6.23　四次多项式牛顿分形之二

3）设 $f(q)=q^5-(10000,0,0,0)$，图 6.24 中，右面是左面旋转后的图像。

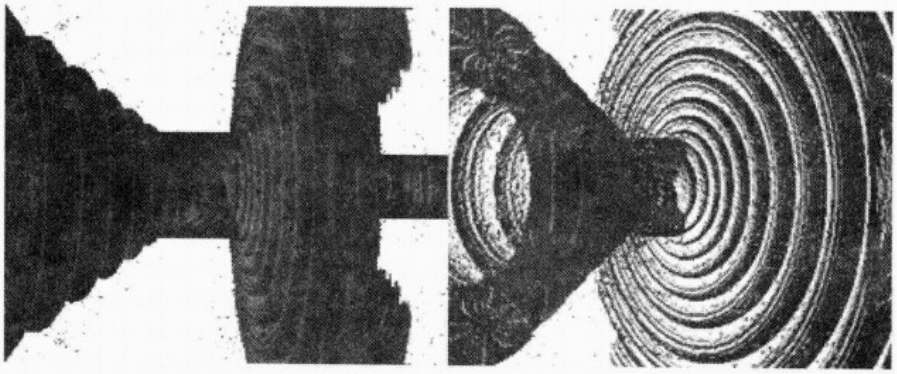


图 6.24　五次多项式牛顿分形

4）设 $f(q)=q^4-q^3-q^2-(10000,0,0,0)$，图 6.25 是在不同定义域里绘制的图形。

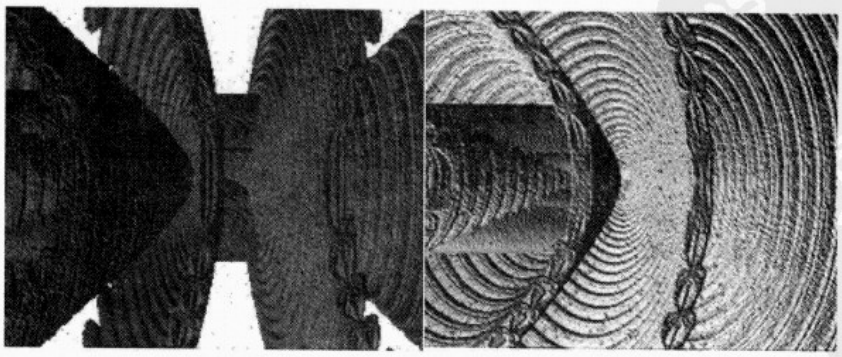


图 6.25　四次多项式牛顿分形之三

6.3.5　四元数 λ 分形

四元数 λ 分形的计算公式仍然是

$$q_{n+1}=\lambda q_n(1-q_n),\quad n=0,1,2,3,\cdots \tag{6.22}$$

式中 λ 是四元数。

它的程序设计可从程序 6.1 得到，由于它跟 Julia 分形非常类似，所以只需要在原来的程序中，把 calc_1 函数做如下改动即可，其他都不用动。

```
double calc_1(double x, double y, double z)
{
  Quaternion<double> q(x, y, z, 0), one(1,0,0,0), lambda(cr,ci,cj,ck),
t;
  double lengde;
  int m=0;
  do {
     t=one;
        t-=q;
     q*=t;
        q*=lambda;

     m++;
 lengde=q.norm();
  } while ((m<iter) && (lengde<2));
  return lengde;
}
```

令 $\lambda=(3.-0.4,-0.1,0)$，得到图 6.26。

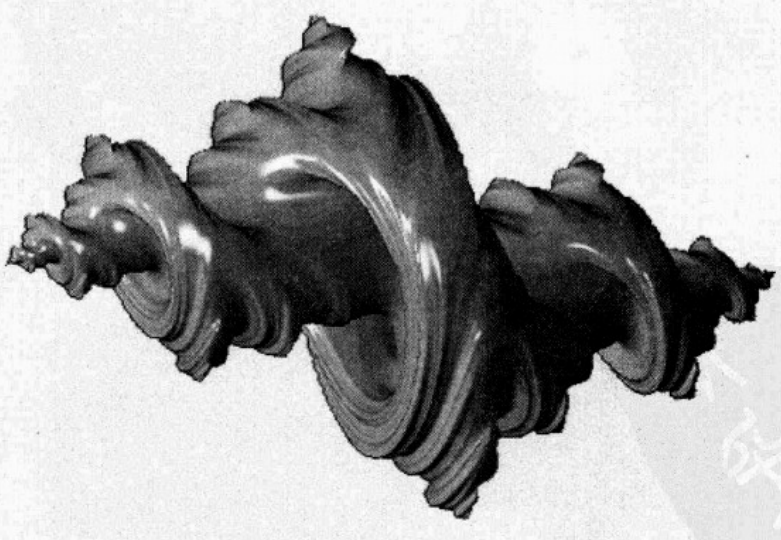


图 6.26　四元数 λ 分形之一

令 $\lambda_1=$（2, 0.6, 0.85, 0.65），$\lambda_2=$（1.2, -0.2, -0.85, 0.6），得到图 6.27 的左右两图。

令 $\lambda_3=$（3, -0.5, 0.2, 0），$\lambda_4=$（1,1,0,0），则有图 6.28 的左右两图。

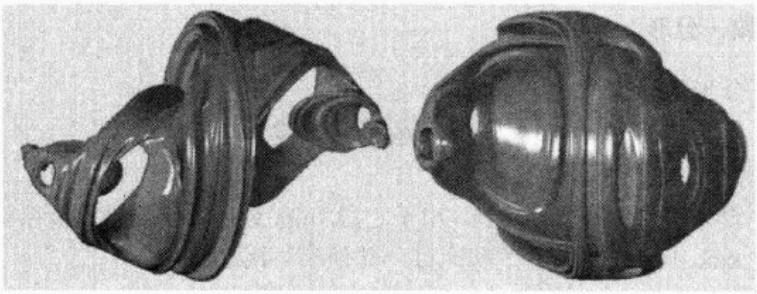


图 6.27　四元数 λ 分形之二

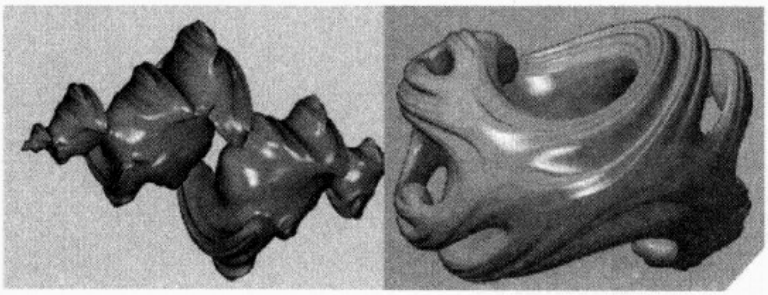


图 6.28　四元数 λ 分形之三

令 λ_5 =（2.92，−0.6，0.2，0.2），λ_6 =（1.2，1.2，0.1，0.1），得到图 6.29 的两个图形。

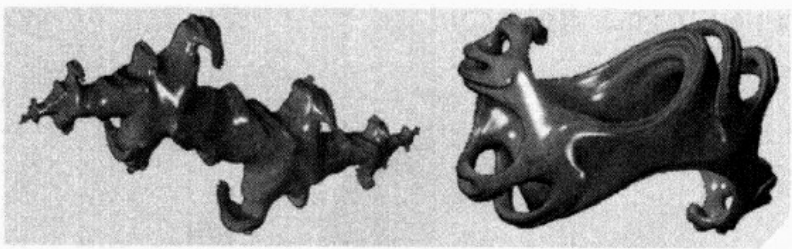


图 6.29　四元数 λ 分形之四

令 λ_8 =（0.555, 0.555, 0.555, 0.555），λ_9 =（0.666, 0.73, 0.666, −0.2），得到图 6.30。

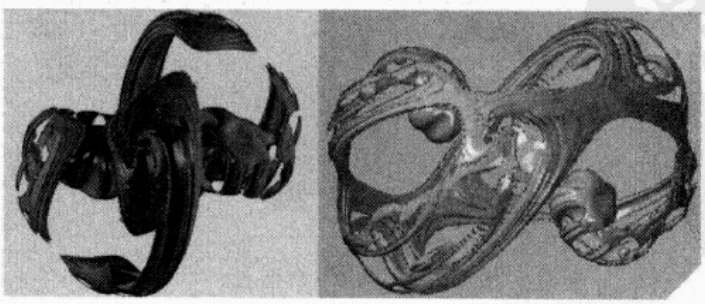


图 6.30　四元数 λ 分形之五

令 λ_7 =（3, −0.5, 0.1, 0），见彩色图片 7（右）。

第 7 章　分形山地自然风景

分形几何学可以说是大自然几何学。大自然中的多数景物和基本几何形状往往具有统计意义上的自相似结构。绵亘的群山叠嶂，蜿蜒的漫长海岸，瞬间的霹雳闪电，神秘的原始森林等等，无不具有分形的特征，且它们的几何维数不是整数。图 7.1 的四幅图像均来自于大自然，难道它们不是分形吗？

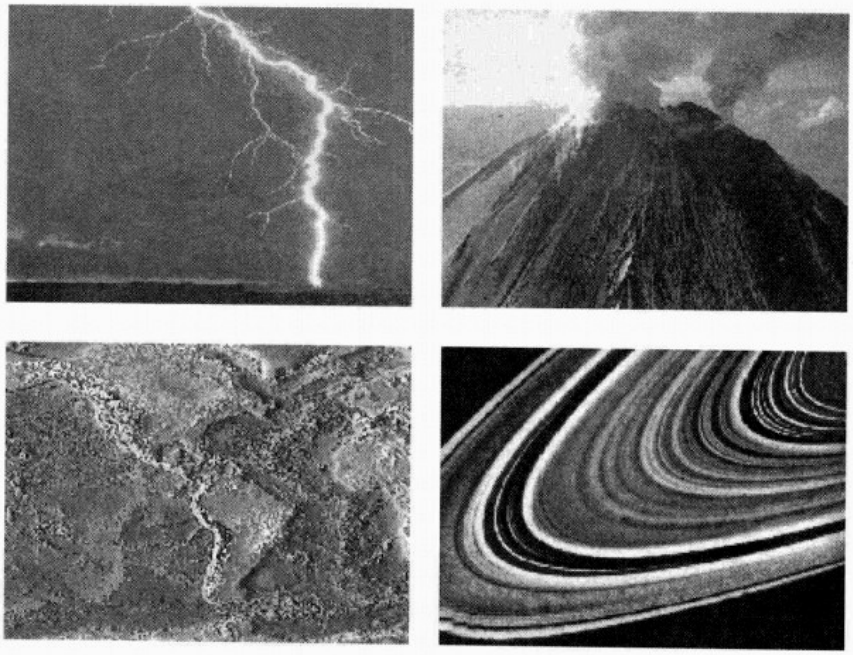


图 7.1　大自然中的分形

自从曼德勃罗创立分形几何学以来，便吸引许多人运用分形方法去描绘大自然的诱人景色，肇始者是曼德勃罗的朋友 R.F.Voss。时至今日，该技术已日臻完美，但其编程难度也越来越大。

7.1　布朗运动与随机中点位移法

我们知道，随机性处处存在，它是自然界的固有性质。即使是所谓“完美晶体”，也有许多杂质原子和晶体缺陷处于随机位置上。随机行走或布朗运动（Brownian motion）过程在物理、化学和生物学中非常重要。

1827 年，英国植物学家布朗在显微镜下第一次观察到水中一颗微粒在不断地做随机（无规则）运动，这是一颗落入液体的花粉在大量液体分子的无规则碰撞下所表现出来的运动状态。从分子运动论的观点看，在任何一瞬间，液体分子从各个方向对花粉微粒

发生撞击。由于涨落现象的存在决定了各个方面的撞击是不平衡的。假定在某一瞬间某个方向的撞击作用较强，则花粉微粒将沿着碰撞方向运动，而在下一个瞬间，将又会朝另一个碰撞作用方向较强的方向行走。这样，花粉在液体中将走一条无规则的曲折路径。

布朗粒子非常微小，其直径约为 10^{-4} cm。它受到周围液体分子碰撞的力只要足够大，就会有微粒行走，粒子越小，布朗运动越显著。通常每个布朗微粒在液体中受周围分子碰撞的频率为每秒 10^{21} 次，在气体中也高达每秒 10^{15} 次。在如此频繁碰撞下，粒子的瞬时运动是无法观察的，人们所能观察到的只是宏观短的时间内的微粒平均运动。图 7.2 显示在一个规定的时间内用不同的时间分隔即用不同的行走步长，布朗运动所走过的路径长度。显然，当时间间隔趋于无穷小时，路径长度将趋于无穷大。

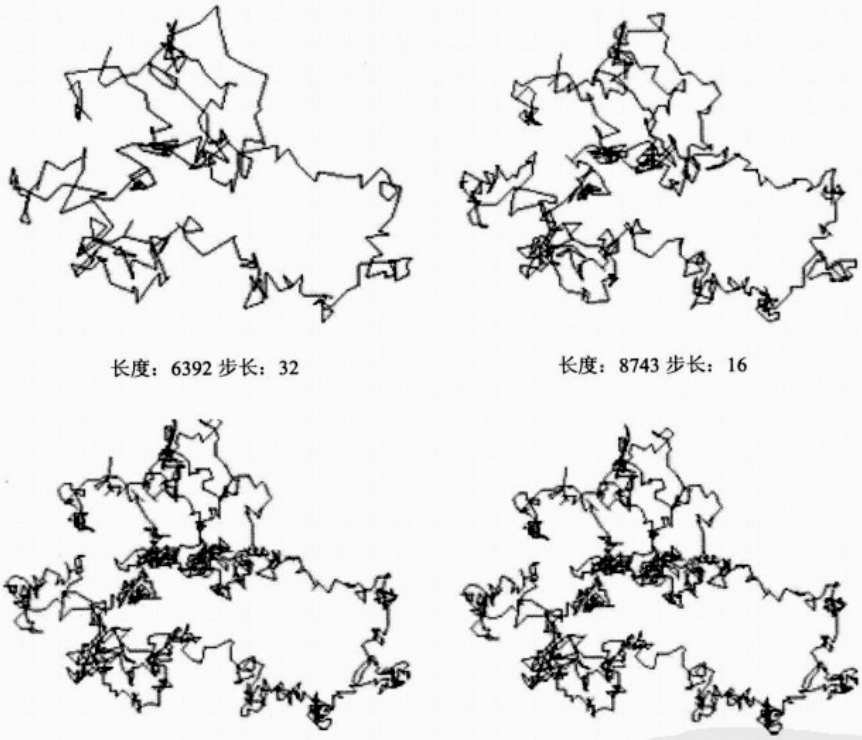


图 7.2　布朗运动的路径

现在我们只把布朗运动限制在一维状态，即让它沿着 x 轴移动或只受来自左或右的碰撞。作为更进一步的简化，假定微粒每次被碰撞后只移动固定位移 s 并且每时刻只有一次碰撞。微粒最初位于 x 轴的原点，它的左碰右撞的概率是相同的。

现在来估计一下它与原点的平均偏差，即微粒位置的方差。如果它被观察到碰撞的次数是 n，则其方差为 ns^2。假定微粒以不变的速率被碰撞，则其 t 时刻的方差与 ts^2 成正比，所以说微粒的标准偏差是 $\sqrt{ts}$。

现用随机变量 X 代表微粒的位置，把分子撞击微粒看作随机变量 L 的一个样本，这

里 L 只能取两个值：+1（左撞）或 -1（右撞），于是有

$$X(n)=\sum_{i=1}^{n}L_i \tag{7.1}$$

由于每个 L_i 是相互独立和同一分布的，所以 $X(n)$ 的平均值为零。每个 L_i 的方差是 s^2，X 的方差是 ns^2，X 是高斯分布的。简单而言就是，给定变量 X 在某个特定开始时间的初值，那么它在任意下一时刻的位置是个随机变量，其均值就是初始位置，其方差与其离初始位置的时间成正比，即

$$Var(X(t_2)-X(t_1))=k\,|t_2-t_1|\,s^2=|t_2-t_1|\,\sigma^2 \tag{7.2}$$

式中 *Var* 代表方差。

为方便起见，令 $t=0$ 开始，到 $t=1$ 为止。而高斯分布的方差与时间成正比($\sigma^2=1$)，故可以以均值为 0，标准偏差为 1 采样高斯随机变量，构造出 $t=1$ 时微粒的位置，从而有 $x(1)=x(0)+N(0,1)$。这里 $N(0,1)$ 是高斯随机变量的一个样本。再令 $t=1/2$，则由 $x(1/2)$ 应满足式（7.2），故让

$$x(1/2)=(x(0)+x(1))/2+N(0,1/\sqrt{2}) \tag{7.3}$$

就可以了。这时 $x(1/2)$ 的位置就是两个端点值的平均位置。这里 $N(0,\ 1/\mathrm{sqrt}(2))$ 是高斯随机变量，其均值为 0，方差为 1/2。

现在我们可以利用简单的递归来产生曲线的其他点了。把[0,1]分成[0, 1/2]和[1/2, 1]两半，然后在它们上面做同样的操作就行：先找出时间间隔的中点，计算两个端点值的平均值，然后从均值为 0 的 Gauss 随机变量中取一个样本，对每次递归，方差乘以 1/2。这种产生间接点的方法称为随机中点位移法（random midpoint displacement），又称 RMD 法，参见图 7.3。

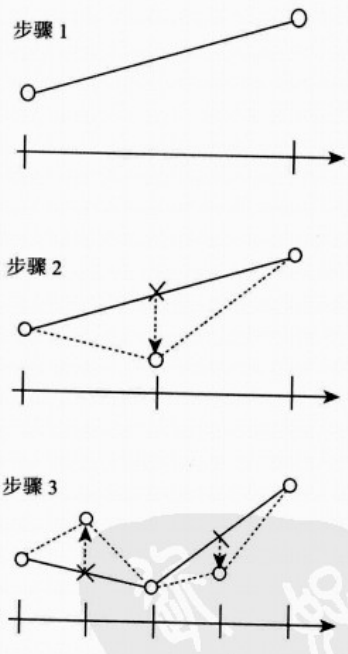


图 7.3　随机中点位移法

随机中点位移法是描画分形风景（fractal landscape）要用到的基本技术，是 N. Wiener 在 1920 年用于研究布朗运动采用的。

随机中点法的计算要点：

1）取当前区间并将它分成两半；

2）利用公式

$$y=\frac{1}{2}(y_1+y_2)+N(0,s)=\frac{1}{2}(y_1+y_2)+sN(0,1) \tag{7.4}$$

计算中间值。

3）在新的区间重复上述过程，把新区间的标准偏差减至原值的 1/2。

如此计算出来的结果可参见图 7.4。

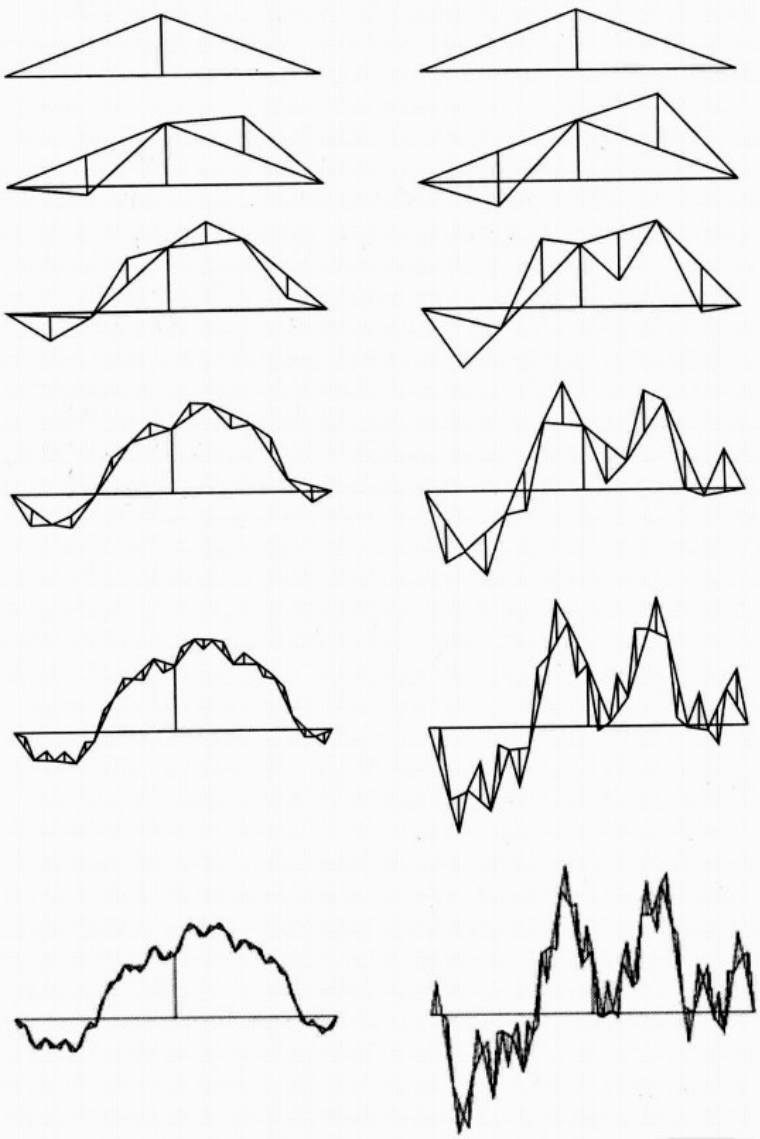


图 7.4　两种 fBM 曲线

可以利用更多的一些参数使过程更细节化。有几种途径可以利用。首先必须确定初始比例因子，即对应于公式（7.2）中的σ^2。这个参数设定了垂直方向的比例。然后，标准偏差在每次细分区间时以$1/\sqrt{2}$比例缩小。如果使用不同的比例因子，从式（7.2）可看出，方差不是严格分半的。这种变化会导致不同类型的曲线，称为布郎运动曲线或fBM（fractional Brownian Motion，分数布朗运动）曲线。见图 7.4。

标度（scale）通常由 H，即赫斯特（Hurst，一个长期在尼罗河水坝工作过的美国人，在 20 世纪 40 年代全面研究了有偏随机游动）指数来参数化的：

$$\text{scale} = \frac{1}{2^H} \tag{7.5}$$

这时，式（7.4）变成了

$$y=\frac{1}{2}(y_1+y_2)+c_i(rand-\frac{1}{2}) \tag{7.6}$$

其中 $c_i=c_{i-1}/2^H$，c_0 是初始值。*rand* 为（0，1）间的随机数。

赫斯特指数也是生成分形曲线维数 D 的度量

$$D=2-H \tag{7.7}$$

当 $H>0.5$ 时，曲线比正常的布朗运动曲线光滑一些（纯布朗运动时，$H=0.5$），这是因为每次细分时，方差也变小了。当 H 趋向于 0 时，曲线变得粗糙，参见图 7.5～图 7.7。

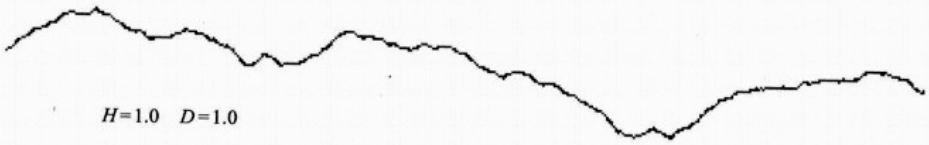


图 7.5　H 与曲线弯曲度之一



图 7.6　H 与曲线弯曲度之二

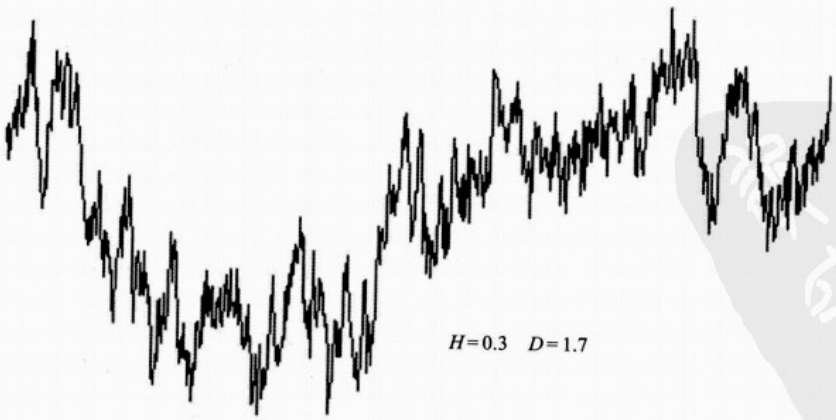


图 7.7　H 与曲线弯曲度之三

7.2　分形山地风景的基本算法

7.2.1　矩形域 RMD 剖分法

这是一种颇为流行的分形山地图画法。

首先给定一个矩形域（见图 7.8 左方），然后连接各边的中点得到一个田字形的四个子块区域（见图 7.8），在田字形的 A, B, C, D, E 五个点（见图 7.9），施行 RMD 算法，得到第一个凸峰（见图 7.8）。具体的 RMD 算法将在下面介绍。

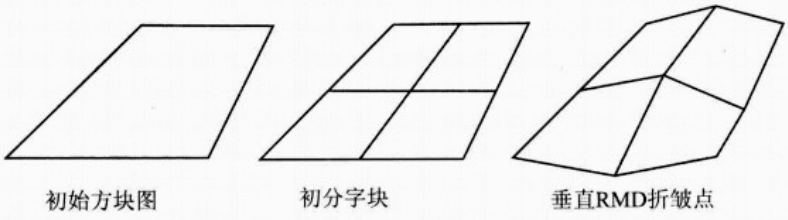


图 7.8　矩形域分割图

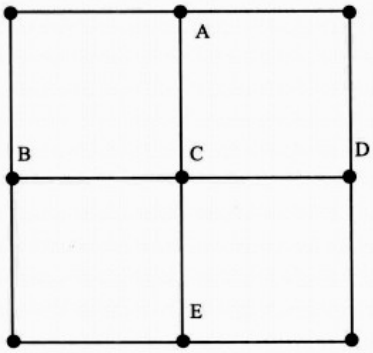


图 7.9　矩形域的 RMD 点

算法的第二步是将田字形再分成四个等同大小的田字形，即在原矩形上形成 4×4 的网格，并在相应的位置施行 RMD 算法。如此进行下去，得到 8×8，16×16，32×32，64×64 等一系列方形网格，并按图 7.9 所示位置施行 RMD 算法。具体进程图可见图 7.10 和图 7.11。

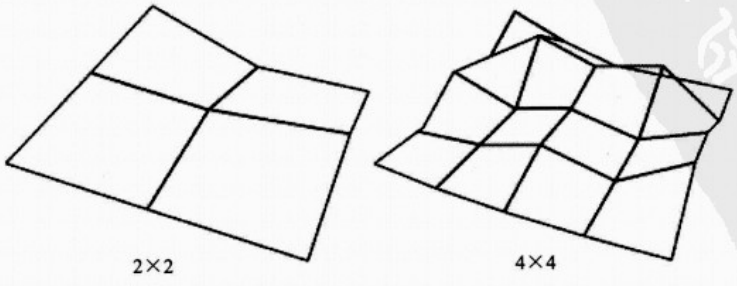


图 7.10　矩形域 RMD 剖分之一

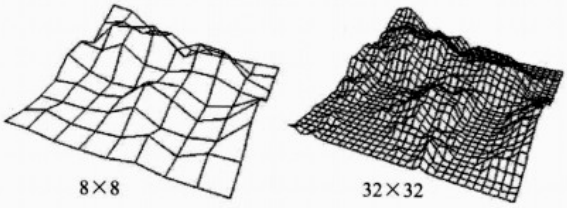


图 7.11　矩形域 RMD 剖分之二

从上列图中可以看出，小迭代数形成的图形较大迭代数形成的图形的中点位移较大。换言之，其局部山峰要高或者说其局部谷底要深。迭代过程中，中点位移量是随机的，但却随 $1/2^H$ 的比率减小。H 与曲面的粗糙度紧密相关。

7.2.2　影响作图的若干因素

网格。方块网格的构造可分两个精细步骤：首先，每个中点的曲面高度是通过最相邻的四个点的平均高度再加一个随机修正量而得到的，见图 7.12 右方中心点或图 7.9 的 C 点。接下来要计算另外中点的高度。显然，当第一步结束时，其余中点（见图 7.9 的 A，B，D，E 点）在该方格只有三个相邻点，要按第一步中 C 点的做法，只有再细分（迭代）网格才行，即迭代数 K=2，见图 7.13。这样，至第 K 次迭代后共有 $2^K \times 2^K$ 个小方格，其每边的锥点数共有 2^K+1 个或写成部分级数和

$$2+(2^0+2^1+\cdots+2^{K-1})$$

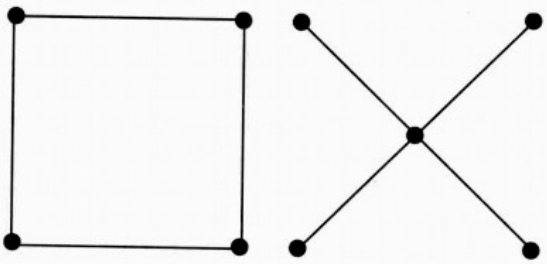


图 7.12　四邻平均加随机修正

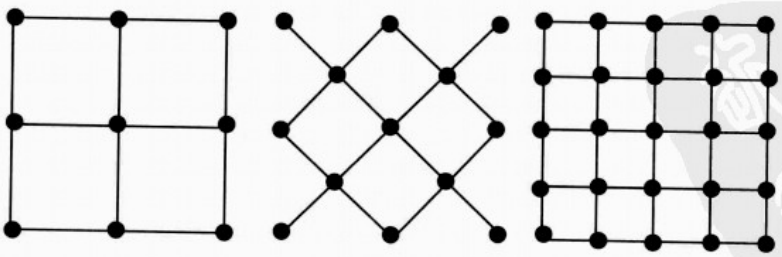


图 7.13　矩形网格之迭代

在第 K 次迭代后，共有 2^{2K} 个小方子块。较大的迭代数意味着较小的网格方块。迭代延续进行使得网格子块愈来愈稠密，且山峰愈来愈高。图 7.14 自上而下是 K=4 和 5 时产生的图像，图 7.15 是 K=6 和 7 时产生的图像。

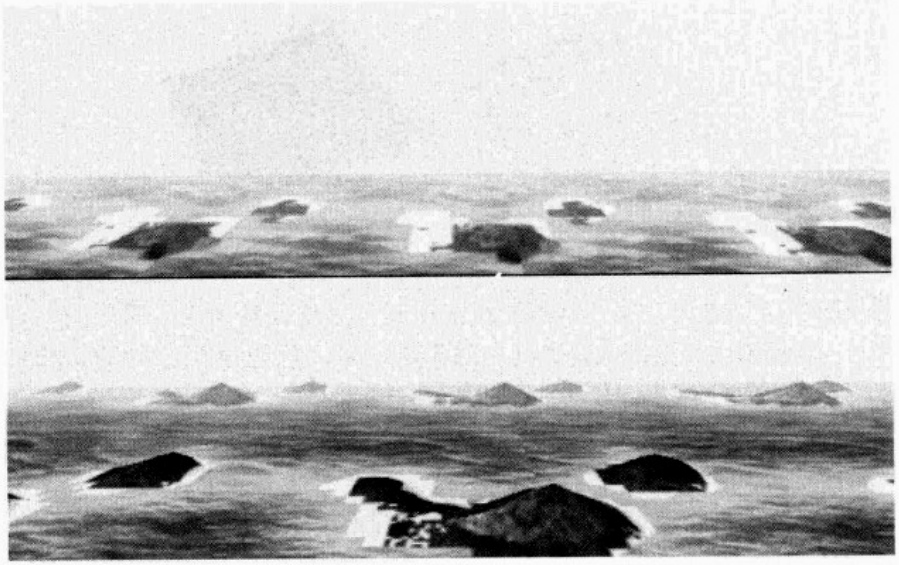


图 7.14　迭代数为 4 和 5 时的图像

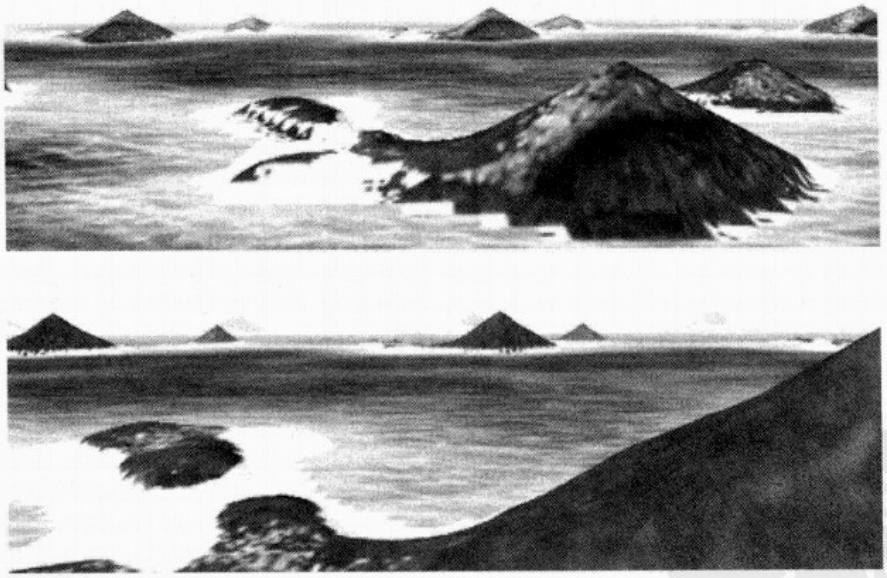


图 7.15　迭代数为 6 和 7 时的图像

随机数。计算机是通过“随机发生器”产生随机数的，其输入被称为“种子（seed）”，输出即为随机数。对同一个种子，将产生同一的随机数，从而也产生同一的山地风景图，否则，会画出不同的图形。一般来说，随机数依赖于正态分布函数，即其均值 $\mu=0$ 和方差 $\sigma^2=1$。不同的分布函数或不同的均值或不同的方差将产生不同的随机数，当然也有不同的随机山地风景。图 7.14～图 7.15 的“种子”为 4，图 7.16 和图 7.17 分别为 5 和 9，而其他参数全部相同。



图 7.16　不同随机数“种子”产生的图像之一

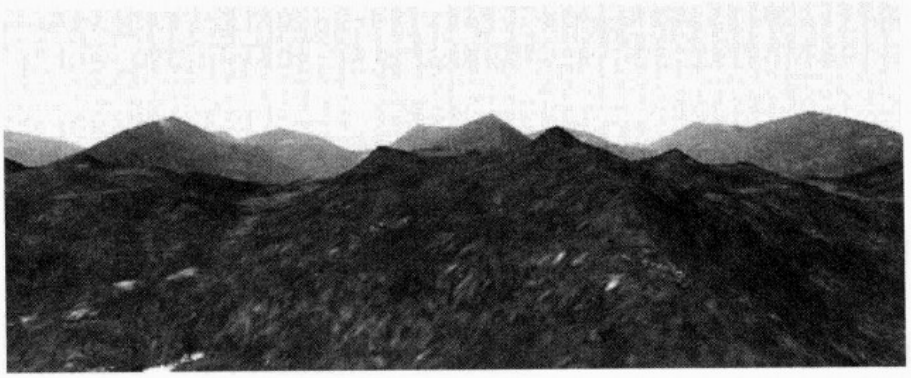


图 7.17　不同随机数“种子”产生的图像之二

粗糙度（roughness）与标度（scale）。粗糙度是一个影响因子，在每次迭代过程中，它直接减少曲面的折皱度。通常，默认的是 2 的因子。一个较大的粗糙度值会产生较光滑的曲面。反之，较小的粗糙度值会勾画出较粗糙的曲面。粗糙度与标度关系如下：

$$scale=1/2^{roughness}$$

于是，大的标度意味着较小的粗糙度，同时得到较粗糙的画面，反之亦然。图 7.18 和图 7.19 分别对应着在其他条件相同时 roughness=0.4 和 0.2 的图像。

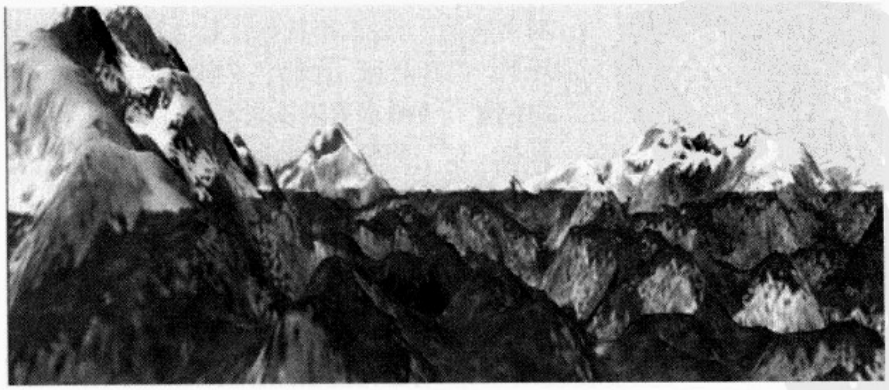


图 7.18　粗糙度为 0.4 的山地图

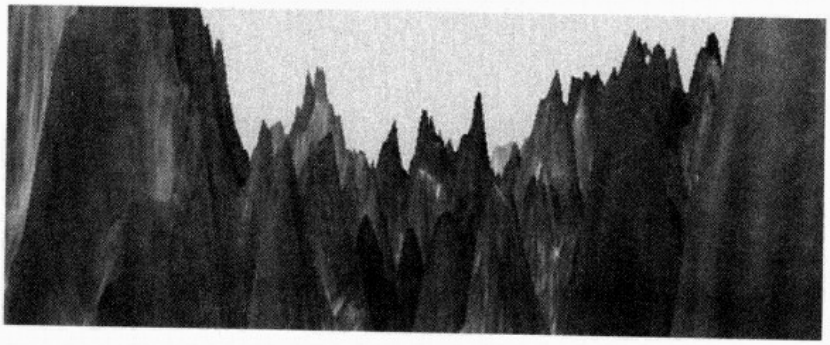


图 7.19　粗糙度为 0.2 的山地图

一般来说，粗糙度用赫斯特指数表示，即公式（7.5）。

网格类型。通用的是较简单的矩形网格，但也可以采用其他形式的网格。图 7.20 从左到右显示了六角型网格和三角形网格的迭代进程。图中的小圆圈代表中点。

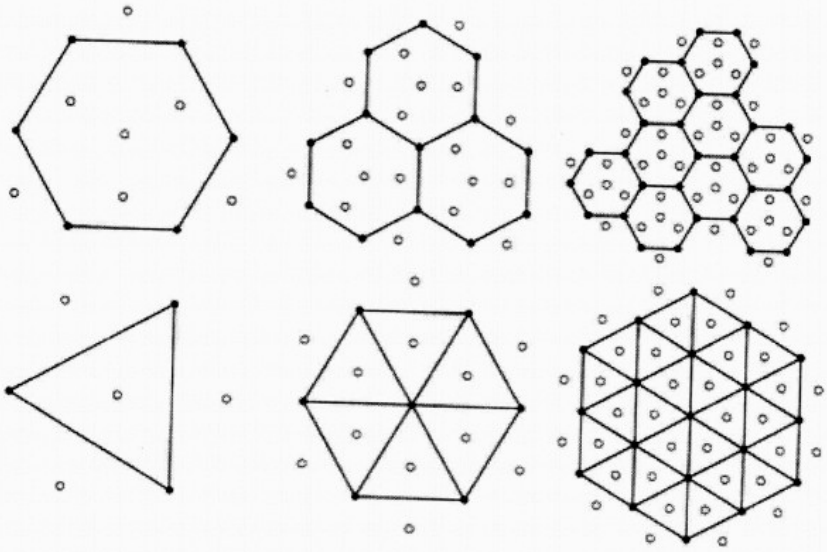


图 7.20　六角形网格和三角形网格的迭代进程

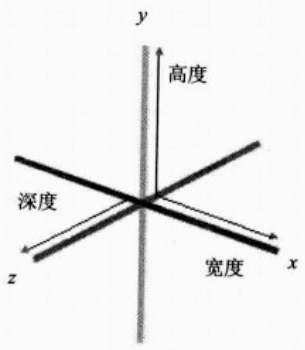


图 7.21　3D 坐标

颜色坡道。山地的形状特点是由基于高度的各种颜色来表现的，但有两到三种颜色是特指的：比如蓝色表示海洋，白色被画于峰顶附近以表示积雪。

坐标。对一个在计算机上的 3D 对象来说，它应该有三个坐标朝向。x 轴正方向为宽度，y 轴正方向为高度，z 轴正方向为厚度。这样，3D 对象的宽与高正对应于计算机屏幕的 x-y 平面，x-z 平面落在方形网格的定位方向，垂直位移在 y 方向，如图 7.21 所示。

海平面。一个完整的山地风景画程序最好有海平面的调节。

图 7.22 和图 7.23 是同一条件下，不同海平面的山地图。

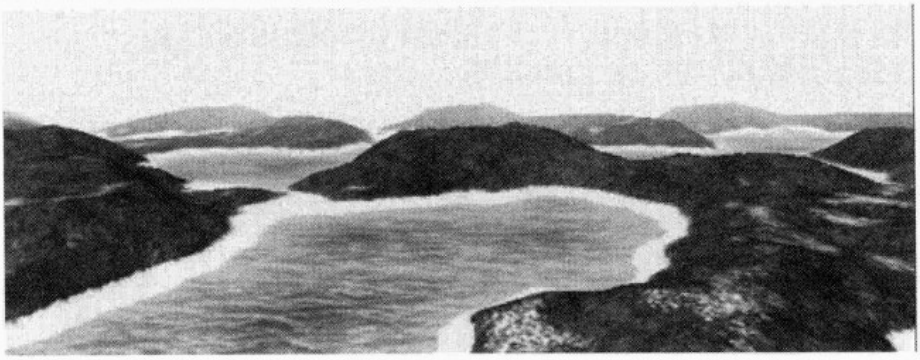


图 7.22　高海平面山地图（低透明度）

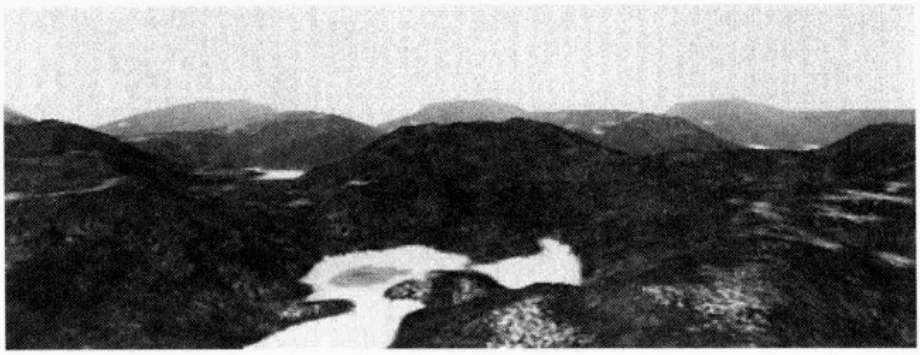


图 7.23　低海平面山地图



图 7.24　$D=2.15$ 的山地

维数。在公式（7.5）的条件下，分形山地的几何维数是 $D=3-H$。

图 7.24 的维数 $D=2.15$，图 7.25 的维数 $D=2.5$。

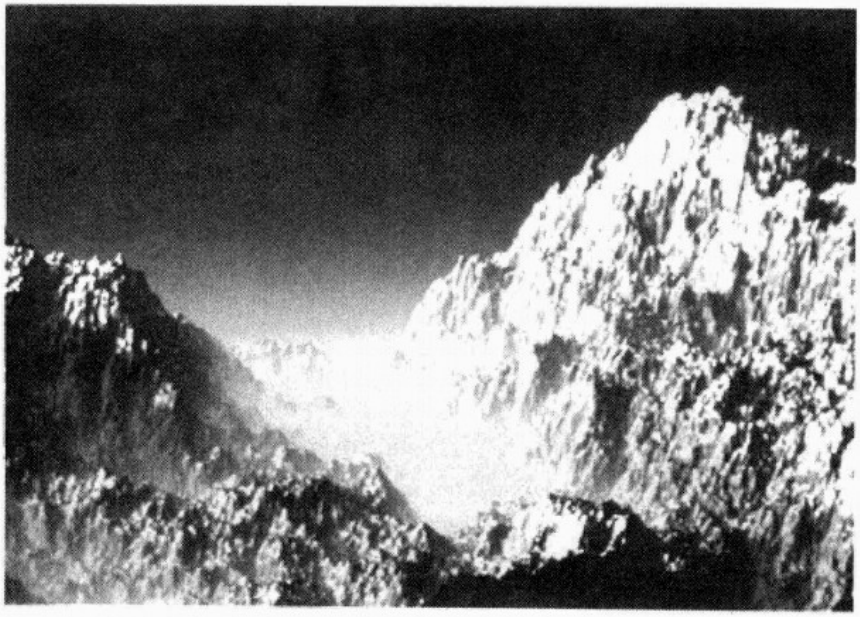


图 7.25　$D=2.5$ 的山地

当今地球上大部分山地的维数是 2.15。

透明度。水的透明度是通过其颜色来表示的。新疆阿尔泰山的卡纳斯湖水由于含有较多的矿物质而显蓝色，而天山的天池湖水则清澈得多而碧绿。图 7.26 和图 7.22 是不同透明度的两幅画。图 7.22 的透明度较低。

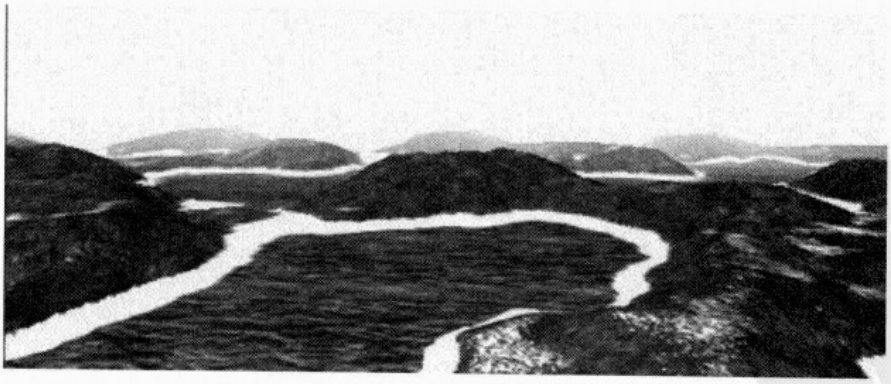


图 7.26　高透明度山地图

对于画出的山地实景来说，通常有两种计算机显示模式：直面型和网状框架型。直面型更接近于自然景色，它是由不同颜色组成的较光滑曲面。网状框架型仿如一张网罩在整个山地上，它最适合于显示每次迭代的框架结构。两种类型的图形可见图 7.27 和图 7.28。

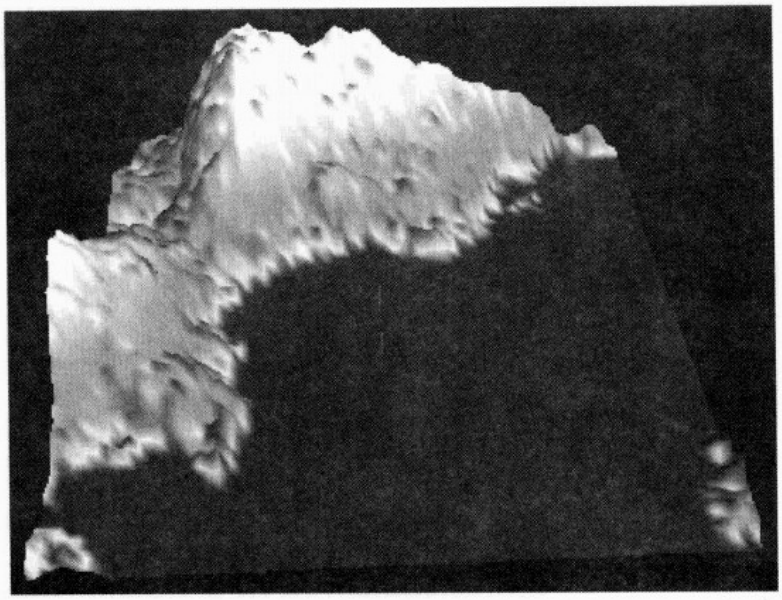


图 7.27　直面型山地风景

在图 7.27 和图 7.28 中，设计的宽度和厚度均为 30，最高高度为 10。迭代次数为 6，即最后网格为 64×64。粗糙度为 0.5。

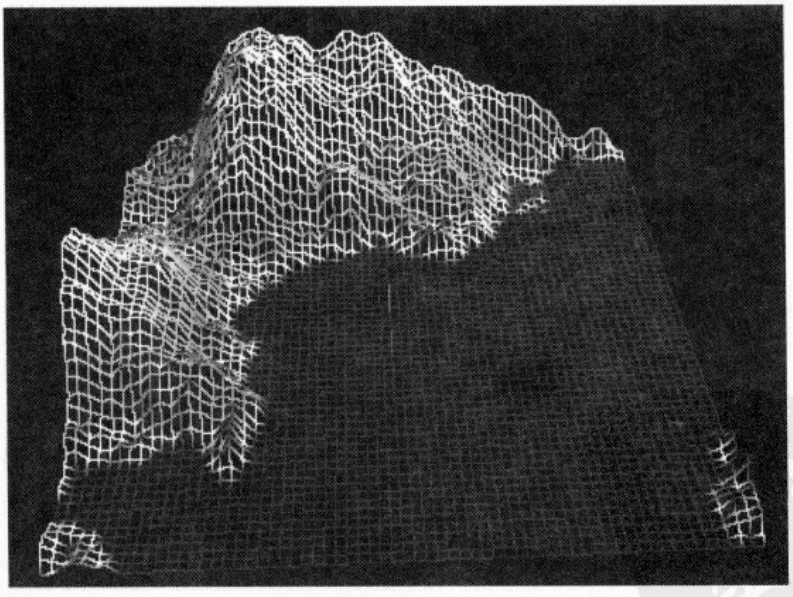


图 7.28　网状框架型山地风景

7.3　随机钻石-方块算法

在 1986 年的 SIGGRAPH 活动中，Gavin S.P.Miller 发表了题为 Definition and Rendering of Terrain Maps 的文章，描述了几个生成分形地形的算法，尤其介绍了钻石-

方块（diamond-square）算法。

7.3.1　一维中点变换

后边要讲的钻石-方块算法是在二维上使用的一种中点变换算法．为此我们先看一维情况。这个所谓的一维中点变换，其实和前面描述的 RMD 算法是很相似的。只不过这个地方更加的具体。

当山脉出现在远处地平线处时，一维中点变换是绘制山脊的好算法。它的工作方法是这样的：

```
以一条水平地平线段开始
重复足够多次{
  对场景中的每条线段做{
    找到线段的中点，
    在 Y 方向上随机移动中点一段距离，
    减小随机数取值范围。
    }
}
```

将随机数值域减小的速度取决于我们想要分形的陡峭程度，每次循环减少得越多，所得山脊线就越平滑。但如果减得太多，则会有明显的锯齿感。我们可以把粗糙度储存在一个常量里，即赫斯特常数。

来看个例子，若以一条 x 从–1.0 到 1.0，y 均为 0 的线段开始，并将随机值范围设为 –1.0 到 1.0（可任意取），这样我们就可在此范围里生成一个数字，并将中点按这个数字移动，就得到了如图 7.29 所示的效果。



图 7.29　一维中点变换

这次经过外圈循环，我们得到两条线段，长度均为原来的一半。随之令随机值也减半，即–0.5 到 0.5。且两个中点都生成一个这个范围内的随机点，结果如图 7.30 所示。

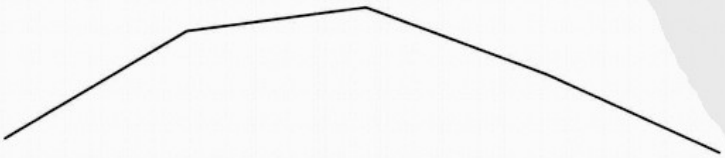


图 7.30　第二次一维中点变换

再次缩减范围，现在是–0.25 到 0.25，再以该范围内的数变换四个中点后，得到了图 7.31。

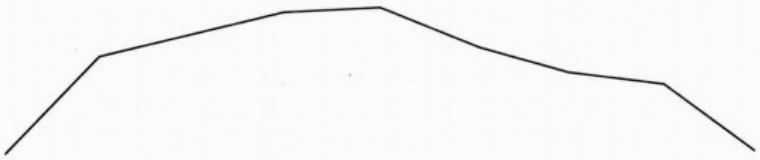


图 7.31　第三次一维中点变换

在此我们注意到两种情况：

首先，它是递归的。实际上，它可以用一个迭代过程很自然地实现。对于这种情况，递归或迭代都行。对于曲面生成代码，使用迭代实现比递归会有一些好处，所以为保持一致，线和面相应的代码都使用迭代实现。

其次，虽然它是个非常简单的算法，但却能创建非常复杂的结果，这正是分形算法的奇妙之处：一些简单的指令可以建立一个具有丰富细节的图像。

很简单地，利用本函数再用一个绘制程序，就得到一维曲线表示的山脉轮廓，如图 7.32 所示。

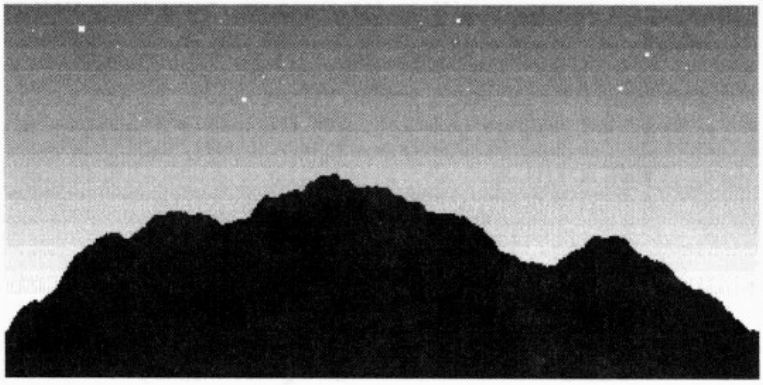


图 7.32　一维曲线表示的山脉轮廓

显然，图 7.32 可以作为窗口景色使用。它的好处是重叠的，如果不介意在每个方向都看见相同的山就可以保留一个小图像，然后拼出整个场景。

在进入 2D 分形曲面之前，再回忆粗糙度常量。这个值决定了每次循环随机数值域的减少量，也就是说，决定分形结果的粗糙程度。使用一个 0.0 到 1.0 之间的浮点数 H（赫斯特常数）。因此 2^{-H} 是 1.0（对于小 H）到 0.5（对大 H）范围内的数且随机数范围在每次循环时都要乘上这个值。如果 H 设为 1.0，则随机数范围将每次循环减半，从而得到一个非常平滑的分形，若将 H 设为 0.0，则范围根本不减小，结果有明显的锯齿感。

图 7.33 是三个山脊，每个用不同的 H 值绘制。



图 7.33　不同 H 值对应的山脊

上边所说的中点变换算法可以用一个一维数组实现，这时数组值表明的却是线段端点垂直位置的高度值，即数值是就是一个一维高度图。它将数据（x 值）映射为高度值（y 值）。

为模拟随机地形，我们可将该算法推广到 3D 空间。为做到这一点，需要一个二维高度值数组，它将索引（x，z）映射为高度（y），数组只需保存高度值（y）。水平面值（x 和 z）可以在分析数组时实时生成。

通过对每个高度指定一个颜色，可以将一幅高度图显示为一幅图像，如图 7.34 所示，高点为白色，低处为黑色。

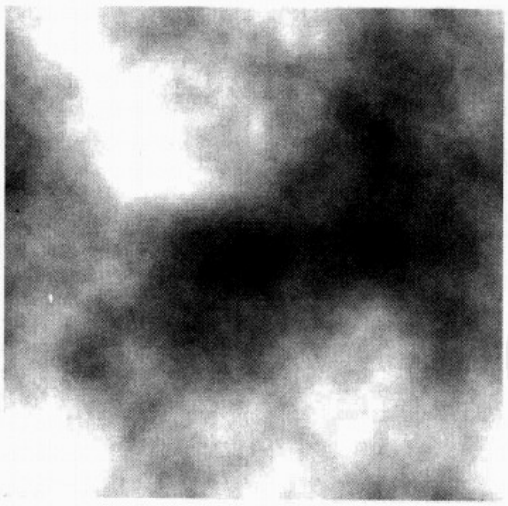


图 7.34　高度图

绘制高度图的方法对于生成云彩纹理图是很有用的，后边还会讨论。

7.3.2　钻石-方块算法

钻石-方块算法可以通过给数组一个“种子”，从而用一个山峰生成一座山脉。比数

组中心点更多的点也得先赋以种子值以构造比较好的结果，其思想如下：先从一个很大的空 2D 数组开始，它应该是方的，维数是 2 的 n 次方加 1（如 33×33，65×65，129×129 等）。将四个角设为相同高度，如果察看所得到东西，它是一个正方形。

取个简单的例子，图 7.35 中用一个 5×5 的数组，图 7.35（a）的四个角种上了初始高度值，表示为黑点．这是递归细分过程的起点，该过程分两步：

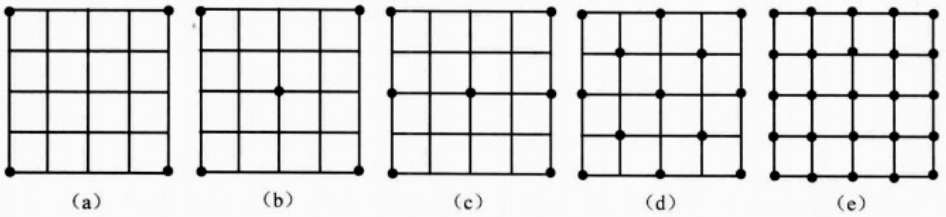


图 7.35　钻石-方块算法示意图

“钻石”步：取四个点的正方形，在正方形中心点生成一个随机值，中心点为两对角线交点，而中心点的值是平均四个角的值再加上一个随机量而计算得到的．这样就得到了一个棱锥，当网格上分布着多个正方形时有点像钻石，见图 7.35（b）和（e）。

方块步：取每个四点形成的钻石，在钻石的中心生成一个随机量，平均这些角的值再加上与钻石步相同的随机量，计算出每条边中点值，这又得到一个正方形，见图 7.35（c）。

这样，如果已经生成了一个种子正方形并经过单独一次细分过程后将得到四个正方形。第二次经过该过程将得到 16 个正方形，第三次得到 64 个正方形。如此而行将使得正方形增长得很快，正方形数目等于 $2^{(I+2)}$，其中 I 为递归细分过程的次数。

参考前五幅插图，图 7.36 示意了使用钻石-方块算法两次后数组发生的情况。

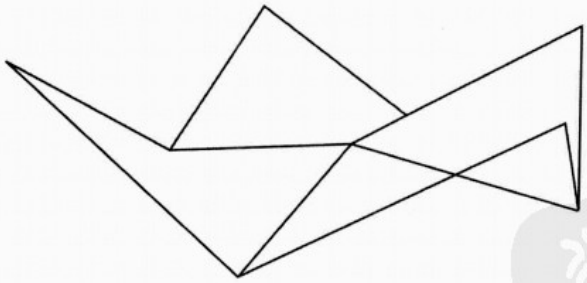


图 7.36　迭代两次后数组的值表示

对于第一遍经过 diamond 步时，我们依据四个角的值在数组中心生成一个值。平均四个角的值（如果种子值相等则完全没必要），并加上一个–1.0 到 1.0 之间的随机值。在图 7.35（b）中，新值显示成黑色，已经存在的点显示为灰色。

对于 square 步，我们在相同的范围内生成随机值。这一步时有四个棱锥（钻石）；他们在数组中心相交，这样我们计算四个 diamond 中心。diamonds 的角被平均以找出新值的基数。图 7.35（c）用黑色显示新值，现存值为灰色。

以上是第一遍，如果用线将这 9 个点编起来，就可以得到一个线框的表面，看起来

就像图 7.36。

现在进行第二遍，再次从 diamond 步开始。第二遍与第一遍有两点不同，首先，我们现在有四个四边形面而不是一个，因此我们得计算四个面的中心；其次，这是关键，生成随机数的范围已经被减小了。作为例子，我们考虑正在使用一个 H=1.0 的值，这将把我们的随机数取值范围将从（-1.0,1.0）减到（-0.5,0.5）在图 7.35（d）中，这步计算得到的四个正方形中心值显示为黑色。

最后，我们进行第二遍的 square 步。有 12 个棱锥中心，故需要计算 12 个新值，如图 7.35（e）中黑色所示。

现在数组中全部 25 个元素都已经生成，可以得到如图 7.37 的线框曲面。

如果分配更大的数组，我们可以进行更多遍，每一遍加入更多细节。例如，5 遍之后表面看起来如图 7.38 所示。

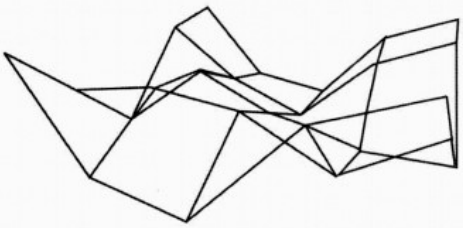


图 7.37　迭代三次后数组的值表示

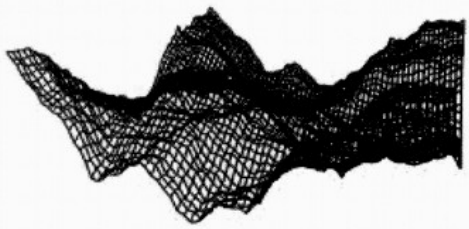


图 7.38　五遍钻石-方块步后的结果

前面提到过，数组维数需要为 2 的整数次方加 1。这是因为 2D 数组中的浮点数个数必须等于 $(2^I+1)^2$。8 次迭代将需要一个 257×257 的浮点数组，对于标准的 32 位 IEEE 浮点数来说超过 256K 内存。

图 7.39 所示的是纹理图的例子。该图经过人工着色，山峰为白色，山谷为绿色，两者之间为灰色。当然也可用其他的配色方案。

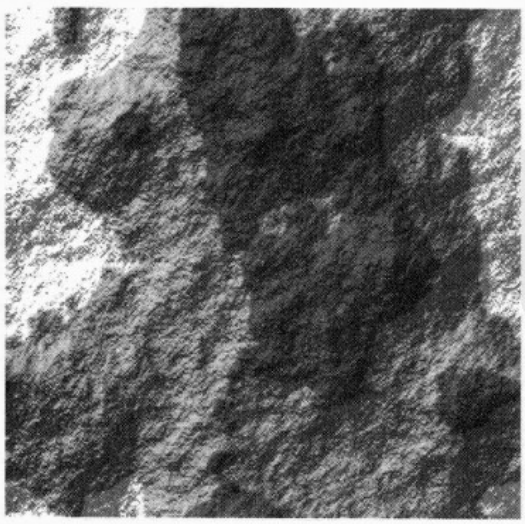


图 7.39　山地俯看图

前面提到过用迭代实现这个程序比递归好。原因是这样的：一个递归实现可以反复采用如下形式：

执行 diamond 步；

执行 square 步；

减小随机数范围；

调用自己四次。

这个过程很简洁，但它要求用不充足的数据生成某些点。为什么呢？经过第一遍之后，将再次调用以执行 square 步，但这时并没有一个棱锥四个角的全部资料。

与之相反，如果用下面这个迭代实现，其伪码如下：

```
当 square 边长度大于 0 时{
遍历数组，对每个正方形表执行 diamond 步
遍历数组，对每个棱锥表执行 diamond 步
减小随机数范围
}
```

这样就消除了递归实现中出现的棱锥角丢失问题。但在生成数组边界的点时还是会碰到这个问题。图 7.40 中，数组中构成棱锥角的位置用亮灰色表示。它们应该被平均以找出新的基本值，即图中黑色的点。

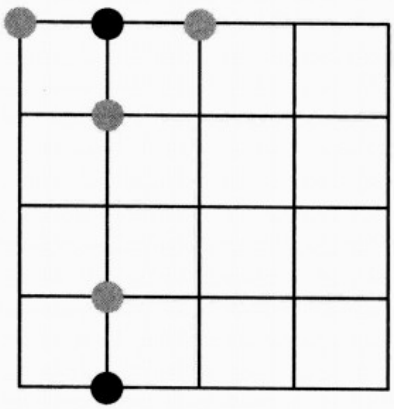


图 7.40　迭代算法示意图

注意用黑色标记的两个值，它们实际上是相同的值。每次在 square 步计算一个边界上的值时，都要同时把它们保存在数组对面边上。

这意味着前面图 7.35（e）中，我们实际上不必计算 12 个单独的值，因为其中的四个在数组相对的两条边上重复，实际上只有 8 个值需要计算。

生成纹理图的程序比较容易。

操作时，可从两次迭代生成的曲面开始，曲面是用线框绘制的，只是简单地将数组中的值用线段连接起来。数组中的值被当作 Y 值，而 X 和 Z 坐标是在数组分析时实时生成的。通过将一个方形分成两个三角形可以轻易地使用三角形绘制出这个曲面。

调节随机种子值，这会导致生成不同的曲面。调高迭代次数值会给曲面增加更多的细节，一般可限制这个值到 10。这样程序在较少内存的计算机上也可以比较顺利的运行。

表面 Hurst 指数控制表面粗糙度，默认值为 0.7，设置较低数值的时候粗糙将非常明显。

diamond-square 算法拼嵌完成的数组非常适合于表示云天的纹理图，与把数组看作一套高度图上的 y 值相反，把它看成云的不透明度数据。最小数组值代表最蓝，天空中最轻的部分；最大的值代表最白，天空中云最重的部分。

分析数组可以生成如图 7.41 所示的纹理图。

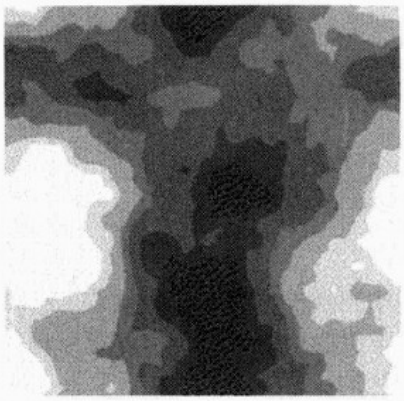


图 7.41　云天纹理图

彩色图片 8 是综合以上分析做出的山地分形图。

7.4　分形山地的 Java 程序设计

在上一节，我们讨论了构造山地分形的钻石（菱形）-方块（正方形）算法，彩色图片 8 是用 VC++编制的程序画成的。这一节将详细地叙述如何用 Java 语言编制程序，叙述中可能会有前面少许内容的重述，但算法却与前面有较大的不同。下面分七个专题讨论并给出程序。这部分资料来源于 Mertin Hughes 撰写的文献。

7.4.1　地形地图与分形地形

地形图是一个将 2D 上的坐标 (x,z) 映射为高度 y 和颜色 c 的函数。换句话说，地形图就是描述一个小范围的地势的函数，见图 7.42。

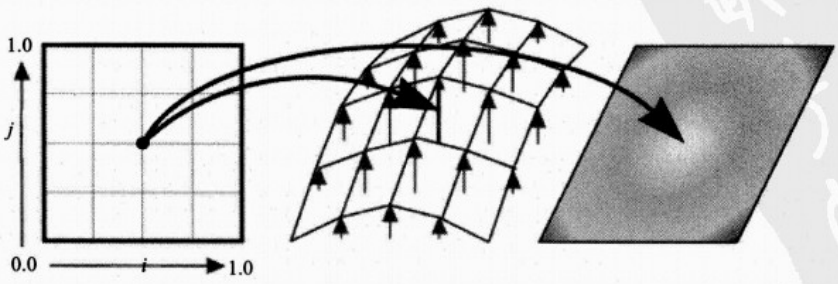


图 7.42　从网格到山地地图

首先定义地形作为一个接口：

```
public interface Terrain {
  public double getAltitude (double i, double j);
  public RGB getColor (double i, double j);
}
```

接着，又假设0.0≤i,j，高度值≤1.0。虽然这并不是必须的。

地形的颜色简单地被描述为 RGB 三元组。为得到其他更加有趣的图像，可以考虑增加其他信息比如表面的亮度等等。然而暂时，下面的类是这样的：

```
public class RGB {
  private double r, g, b;

  public RGB (double r, double g, double b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }

  public RGB add (RGB rgb) {
    return new RGB (r + rgb.r, g + rgb.g, b + rgb.b);
  }

  public RGB subtract (RGB rgb) {
    return new RGB (r - rgb.r, g - rgb.g, b - rgb.b);
  }

  public RGB scale (double scale) {
    return new RGB (r * scale, g * scale, b * scale);
  }

  private int toInt (double value) {
    return (value < 0.0) ? 0 : (value > 1.0) ? 255 :
      (int) (value * 255.0);
  }

  public int toRGB () {
    return (0xff << 24) | (toInt (r) << 16) |
      (toInt (g) << 8) | toInt (b);
  }
}
```

RGB 类定义一个简单的颜色容器。我们提供一些基本的方法，可以进行颜色的运算也可以转换一种浮点颜色成封装的整数格式颜色。

首先看一个由从正弦和余弦函数计算出的抽象的地形，见图 7.43。

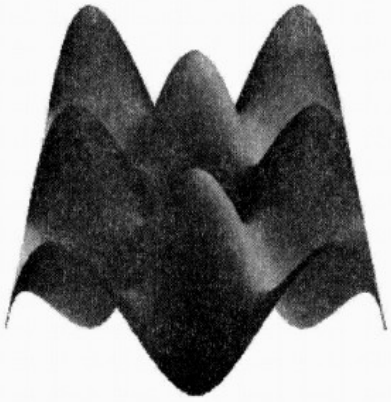


图 7.43　一个抽象的地形图

```
public class TranscendentalTerrain implements Terrain {
  private double alpha, beta;

  public TranscendentalTerrain (double alpha, double beta) {
    this.alpha = alpha;
    this.beta = beta;
  }

  public double getAltitude (double i, double j) {
    return .5 + .5 * Math.sin (i * alpha) * Math.cos (j * beta);
  }

  public RGB getColor (double i, double j) {
    return new RGB (.5 + .5 * Math.sin (i * alpha),
                    .5 - .5 * Math.cos (j * beta), 0.0);
  }
}
```

上面的山地构造器接受定义地形频率的两个参数，它们通过 Math.sin()和 Math.cos()来计算高度和颜色。这些函数返回值在–1 和＋1 之间。

简单的数学地形是没有现实意义的，我们想要的是至少看上去近似真实的东西，那就是分形世界。人们可以使用真正的地势文件作为我们的地形地图，当然这是不难实现的，见图 7.44。

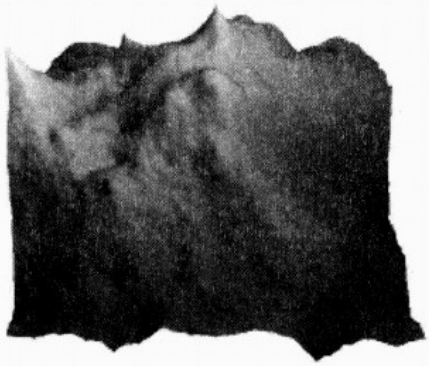


图 7.44　一种分形地形图

分形是可以展现出自相似性的事物（一个函数或物体）。例如，曼德勃罗集是一个分形函数。如果放大曼德勃罗集，将发现类似曼德勃罗集主体的微小的内部结构。山脉也是分形，至少在其外形上表现如此。一座单个山体的局部特点类似山脉的整体特点，甚至细到石头的粗糙程度。我们将按自相似性这个原则产生分形地形，其本质是最初产生一个粗糙的随机地形，然后将递归地增加类似整体结构的随机性细节，不过规模是越来越小。使用的实际算法是菱形算法，最早是由 Fournier、Fussell 和 Carpenter 在 1982 年提出的。

可以通过下面的步骤构建分形地形：

1）首先给网格的四壁角点设定一个随机的高度。

2）平均这四个角落的高度，再增加一个随机的扰动，然后设定为这个网格中点的高度[见图 7.46 的(b)图]。鉴于这种在网格产生菱形图样故称为菱形（钻石）步骤。（开始的迭代，看起来不像菱形是因为它们是在网格的边缘）

3）将得到的每个（钻石）菱形，对四个角落的高度加以平均，再增加一个随机的扰动，这样就设定了这个菱形的中点的高度[见图 7.46(c)]。鉴于在网格产生正方形样式故称为正方形（方块）步骤。

4）继续进行，再对应用正方形步骤生成的每个正方形应用菱形步骤，然后再对在菱形步骤产生的每个菱形应用正方形步骤，反复直到网格充分地密集，见图 7.45。

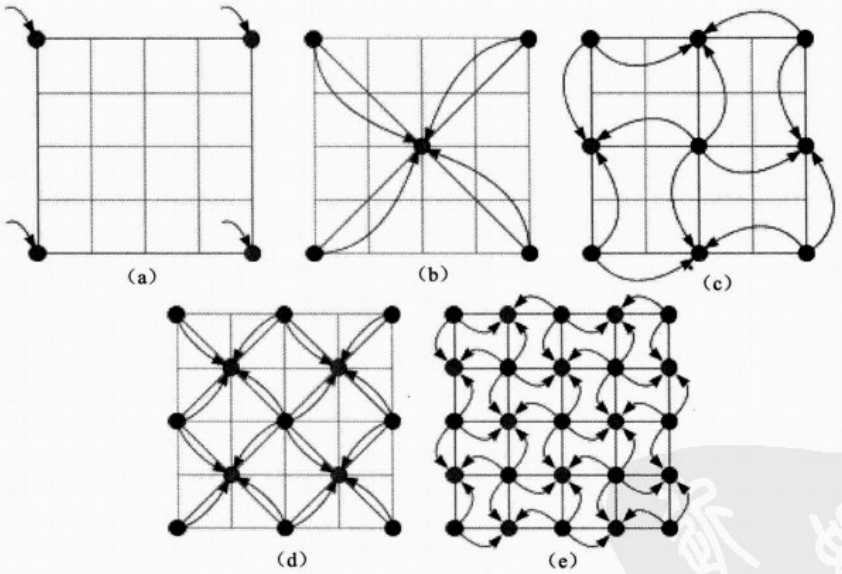


图 7.45　菱形-方块算法

如何确定对网格的扰动？方法是：我们开始让 $0.0 < r$（粗糙系数）< 1.0。在菱形-正方形算法迭代到第 n 次时，增加的随机扰动满足：$-r^n \leqslant$ 扰动 $\leqslant r^n$。从根本上说，人们增加给网格的细节越多，则做变动的量越少。分形的局部的细小变化相似于整体的巨大变化。

如果选择的粗糙系数很小，那么得到的地形将是非常光滑的，变动会非常迅速地减少到零。如果选择一个较大值，那么地形将非常粗糙，因为变动在小网格分裂后依然很

大。图 7.46 是当粗糙度为 0.6 时产生的山地图形。

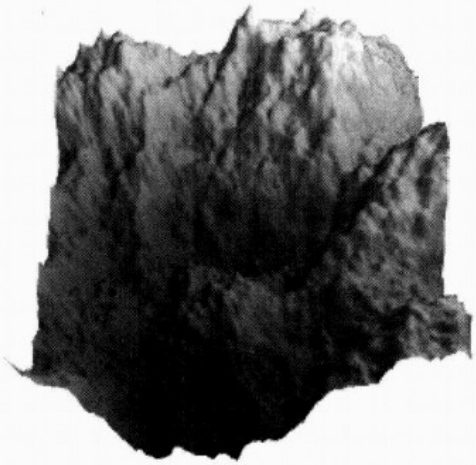


图 7.46　一个粗糙的分形山地

以下是分形地形地图的代码实现：

```
public class FractalTerrain implements Terrain {
  private double[][] terrain;
  private double roughness, min, max;
  private int divisions;
  private Random rng;

  public FractalTerrain (int lod, double roughness) {
    this.roughness = roughness;
    this.divisions = 1 << lod;
    terrain = new double[divisions + 1][divisions + 1];
    rng = new Random ();

    terrain[0][0] = rnd ();
    terrain[0][divisions] = rnd ();
    terrain[divisions][divisions] = rnd ();
    terrain[divisions][0] = rnd ();

    double rough = roughness;
    for (int i = 0; i < lod; ++ i) {
      int q = 1 << i, r = 1 << (lod - i), s = r >> 1;
      for (int j = 0; j < divisions; j += r)
        for (int k = 0; k < divisions; k += r)
          diamond (j, k, r, rough);
      if (s > 0)
        for (int j = 0; j <= divisions; j += s)
          for (int k = (j + s) % r; k <= divisions; k += r)
            square (j - s, k - s, r, rough);
      rough *= roughness;
    }

    min = max = terrain[0][0];
    for (int i = 0; i <= divisions; ++ i)
      for (int j = 0; j <= divisions; ++ j)
```

```
        if (terrain[i][j] < min) min = terrain[i][j];
        else if (terrain[i][j] > max) max = terrain[i][j];
    }

    private void diamond (int x, int y, int side, double scale) {
      if (side > 1) {
        int half = side / 2;
        double avg = (terrain[x][y] + terrain[x + side][y] +
          terrain[x + side][y + side] + terrain[x][y + side]) * 0.25;
        terrain[x + half][y + half] = avg + rnd () * scale;
      }
    }

    private void square (int x, int y, int side, double scale) {
      int half = side / 2;
      double avg = 0.0, sum = 0.0;
      if (x >= 0)
      { avg += terrain[x][y + half]; sum += 1.0; }
      if (y >= 0)
      { avg += terrain[x + half][y]; sum += 1.0; }
      if (x + side <= divisions)
      { avg += terrain[x + side][y + half]; sum += 1.0; }
      if (y + side <= divisions)
      { avg += terrain[x + half][y + side]; sum += 1.0; }
      terrain[x + half][y + half] = avg / sum + rnd () * scale;
    }

    private double rnd () {
      return 2. * rng.nextDouble () - 1.0;
    }

    public double getAltitude (double i, double j) {
      double alt = terrain[(int) (i * divisions)][(int) (j * divisions)];
      return (alt - min) / (max - min);
    }

    private RGB blue = new RGB (0.0, 0.0, 1.0);
    private RGB green = new RGB (0.0, 1.0, 0.0);
    private RGB white = new RGB (1.0, 1.0, 1.0);

    public RGB getColor (double i, double j) {
      double a = getAltitude (i, j);
      if (a < .5)
        return blue.add (green.subtract (blue).scale ((a - 0.0) / 0.5));
      else
        return green.add (white.subtract (green).scale ((a - 0.5) /
  0.5));
      }
    }
```

在构造函数中，同时给定了粗糙系数 roughness 和细节层次 lod。细节层次是指执行迭代的次数，对于细节层次 n，会产生 $(2^n+1)\times(2^n+1)$ 个网格。每次迭代，对网格中的每个正方形应用菱形法则，每个菱形应用正方形法则，然后，计算出最大值和最小值来刻画地形的高度和低谷。

为了计算一个点的高度，我们测量并返回离该点最近的网格样本点。更理想地，似乎应该用周围的样本点插值计算，但不如上面的方法简单，并且已经足够好了。在这个最终的程序中不会出现这个问题，因为程序实际上会匹配地按要求的细则程度定出地形的取样位置。给地形上色时，可以根据抽样点的高度，简单地画上在蓝色、绿色和白色之间的值。

7.4.2　用马赛克技术处理地形

现在已经有一张定义在一个方形领域的地形地图了，接下来需要解决怎样将它实际画到屏幕上去。我们可以向空间射出光线，设法确定会到达哪部分的地形。然而这种方法，是非常慢的。这里改用的方法是将光滑的地形近似以一系列相连的三角形去表现，就是说，将用马赛克去处理地形。

要形成三角网络，我们将均匀地从地形中抽样出规则的网格，然后用三角形去复盖网格中的每个正方形使其分为两个。有许多有趣的技术可以简化这个三角网络，但还要关心速度的快慢。

以下代码是以分形地形数据表现地形网格的基础，并且还要缩减地形的垂直坐标以免高度过大，见图 7.47。

```
double exaggeration = .7;
int lod = 5;
int steps = 1 << lod;
Triple[] map = new Triple[steps + 1][steps + 1];
Triple[] colors = new RGB[steps + 1][steps + 1];
Terrain terrain = new FractalTerrain (lod, .5);
for (int i = 0; i <= steps; ++ i) {
  for (int j = 0; j <= steps; ++ j) {
    double x = 1.0 * i / steps, z = 1.0 * j / steps;
    double altitude = terrain.getAltitude (x, z);
    map[i][j] = new Triple (x, altitude * exaggeration, z);
    colors[i][j] = terrain.getColor (x, z);
  }
}
```

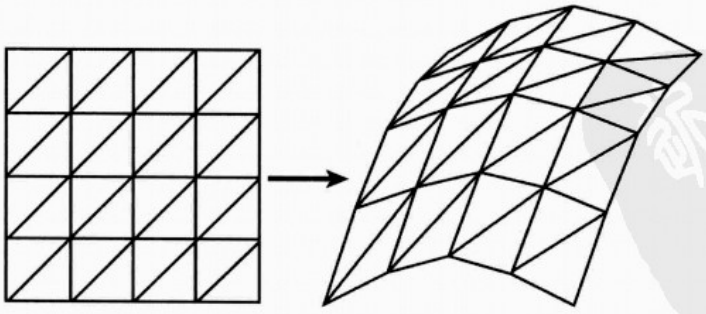


图 7.47　用马赛克技术去处理地形

为什么用三角形而不是正方形？这是因为在空间中任意四点，它们共面的可能性很小。所以在 3D 空间，正方形网格不大可能在一个平面内。如果分解地形为三角形，就可以保证任何三点在空间是共面的，这意味着，画完之后地形中没有间隙。

```
public class Triangle {
  private int[] i = new int[3];
  private int[] j = new int[3];
  private Triple n;
  private RGB[] rgb = new RGB[3];
  private Color color;

  public Triangle (int i0, int j0, int i1, int j1, int i2, int j2)
{
    i[0] = i0;
    i[1] = i1;
    i[2] = i2;
    j[0] = j0;
    j[1] = j1;
    j[2] = j2;
  }
}
```

在地形网格点中，三角形的数据结构以索引 (i, j) 存放各个端点，连同存储的还有以后将填装的法线和颜色信息。这里用以下代码片断创建三角形数组：

```
int numTriangles = (steps * steps * 2);
Triangle[] triangles = new Triangle[numTriangles];

int triangle = 0;
for (int i = 0; i < steps; ++ i) {
  for (int j = 0; j < steps; ++ j) {
    triangles[triangle ++] = new Triangle (i, j, i + 1, j, i, j + 1);
    triangles[triangle ++] = new Triangle (i + 1, j, i + 1, j + 1, i,
j + 1);
  }
}
```

下面来看看怎样把这些三角形在屏幕显示出来，但首先必须要解决光线问题。

7.4.3　光线和阴影

没有经过进行光线和阴影计算的图片看起来会与图 7.48 相似。

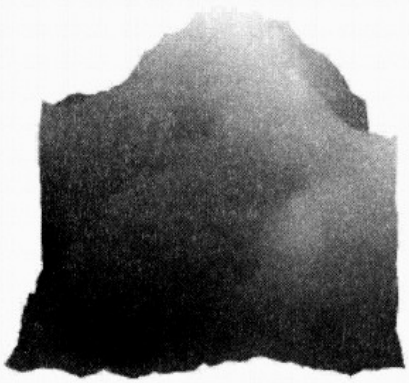


图 7.48　未经光线处理的山地图

可以做两件事来修改它：首先，在环境周围增加分散的光源以使地形中的元素可以以相对太阳的位置发光，然后，将增加阴影以表现虚拟物体可以阻挡虚拟光线的传播。

如果粗糙的表面面对一个光源，反射光的数量与入射角的余弦将成比例。直接照射到表面的光线将被强烈反射，而斜角照射到表面的光线会被微弱地反射。此外，所有的表面都会均匀地被许多周围的散射的光照射，见图 7.49。

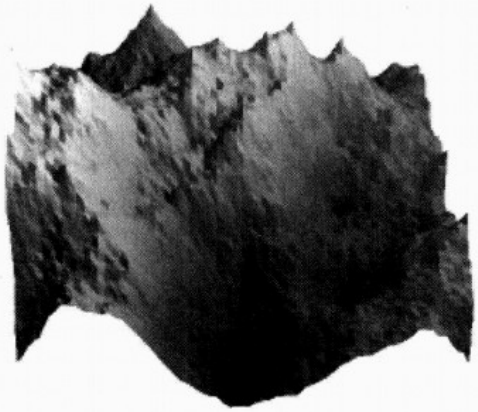


图 7.49　一个漫射照亮的山地

对地形应用这个简单的光照模型时，一种选择是对地形中的每个点计算它是否面对太阳。如果是，入射光的角度是多少？入射角的余弦等于地形法向量和光线方向的点乘。当然，对屏幕上的每个点作计算是有相当代价的，这可以代之以对每个三角形作一次这样的计算，结果就应该相当满意了。将太阳作为一个点，而不用考虑它是地球大小的多少倍并允许它可以被放置在屏幕的任何地方。第一步，计算出太阳到地面的向量：

$$\text{sun: } P_{sun} = (X_{sun}, Y_{sun}, Z_{sun})$$

$$\text{terrain point: } P = (X, Y, Z)$$

$$\text{light vector: } V_{light} = P - P_{sun} = (X - X_{sun}, Y - Y_{sun}, Z - Z_{sun})$$

为帮助计算，可使用一个辅助类：

```
public class Triple {
  private double x, y, z;

  public Triple (double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public Triple add (Triple t) {
    return new Triple (x + t.x, y + t.y, z + t.z);
  }

  public Triple subtract (Triple t) {
    return new Triple (x - t.x, y - t.y, z - t.z);
  }
```

```
    public Triple cross (Triple t) {
      return new Triple (y * t.z - z * t.y, z * t.x - x * t.z,
        x * t.y - y * t.x);
    }

    public double dot (Triple t) {
      return x * t.x + y * t.y + z * t.z;
    }

    public double length2 () {
      return dot (this);
    }

    public Triple normalize () {
      return scale (1.0 / Math.sqrt (length2 ()));
    }

    public Triple scale (double scale) {
      return new Triple (x * scale, y * scale, z * scale);
    }
  }
```

Triple 类提供了空间中一个点或向量，接下来要提供对其进行数学操作的各种方法。注意到这里的信息存储是双精度型的。如果对速度有更多要求，也可以用浮点型。同样，注意到计算中会生成很多临时的 Triple 类，可以（像 Java 3D API 那样）对这些类进行重复应用以加快速度。

再则，我们需要表面的法向量。要计算的就是一个直立在每个三角形上的向量。如果取三角形所在平面上的两个向量（非平行的）进行叉积，就得到了表面法向量。并且，我们就用三角形的两边作为这两个向量：

$$\text{Triangle: } T=[P_0,P_1,P_2]$$

$$\text{Sides: } V_a=(P_1-P_0)=(X_1-X_0,Y_1-Y_0,Z_1-Z_0)$$

$$V_b=(P_2-P_0)=(X_2-X_0,Y_2-Y_0,Z_2-Z_0)$$

$$\text{Surface normal: } n_T=V_a\times V_b=(Y_aZ_b-Z_aY_b,Z_aX_b-X_aZ_b,X_aY_b-Y_aX_b)$$

再其次，要记得光线的强度按比例随旋转正方形与光源的距离增大而减小。由于球形的表面面积是 $4\pi r^2$，于是距太阳距离为为 d 的地方，太阳射出的光会散布到面积为 $2\pi r^2$ 的区域内。对于在地球上的我们，这并没有什么意义，然而现在的场景中，太阳是非常近的，所以我们对这样的计算是关心的，参见图 7.50。

现在来计算地形上一点的照明情况。目前已知的是表面的法向量、光线方向、还有与太阳的距离。光线的反射和光线入射角的余弦成正比，算法就是用两个单位向量的点乘，周围的光线作为常数。于是有：

周围光照（太阳强度）：s

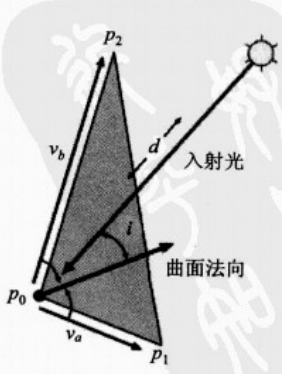


图 7.50　计算漫射照亮表面的光

单位曲面法向量：$n' = n / |n|$

光线距离：$d = |V_{light}|$

单位光向量：$V'_{light} = V / d$

光线入射量：$i = V'_{light} \cdot n'$

面对太阳的照明度：　$(i < 0) : l = a - i \cdot s / d^2$

背对太阳的照明度：　$(i \geq 0) : l = a$

这样，我们就可以计算在地形地图中任意三角形的光线反射量了。

上面讨论中的一个明显的错误就是将三角形单一地着色了，这总有点与我们对地形的定义背道而驰。这里的地形在高度和着色方面是应该是多种色彩且具有连续性。但是目前却用同一种着色于每个三角形，这意味着我们的图片不符合实际情况，即临近的三角形着色互不相同，而好的图像中不应有绝对的颜色断层。为了克服这种两难处境，需要引入一种更好的阴影模型。平滑的阴影意味着每个三角形拥有单独的颜色，但是顾柔（Gouraud）阴影法却允许我们去为每个三角形的每个顶点计算一种颜色，然后让色彩在三角形的内部平滑地过渡，见图 7.51。

这种模型可以轻松地确保连续性。实际上，只要计算地形中每个网格点的颜色，然后让三角形的顶点从网格上的颜色中获取合适的部分。如果两三角形相邻，则它们共享一条边及其上的顶点，故这条边也具有相同的颜色，见图 7.52。

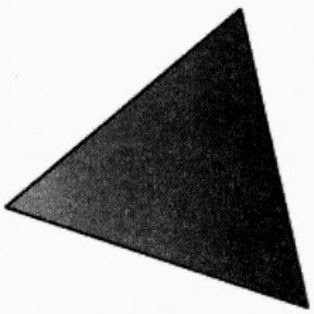


图 7.51　顾柔阴影法得到的三角形

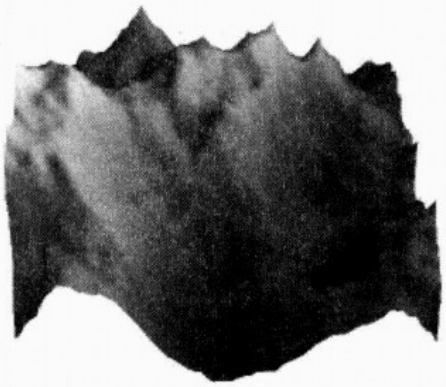


图 7.52　由顾柔阴影法画出的山地图

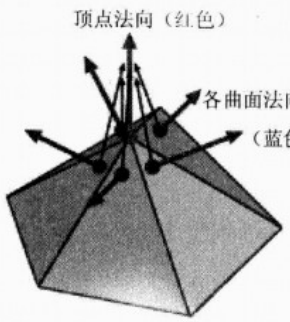


图 7.53　平均共顶点的所有三角形的法向量

使用前面提到的算法计算网格中一点的颜色，所需的额外信息只有地形上顶点的法向量。虽然也可以用多种方法去计算，但不管怎样我们都要计算地形表面的法向量。一种简单的方法是平均共享这个顶点的所有三角形的平面法向量。另外，稍复杂一些的，还可以计算均衡穿过周围顶点的平面的法向量。为简单起间，我们使用第一种方法，见图 7.53。

下面的代码计算三角形数组的法向量和光照信息。

```
double ambient = .3;
double diffuse = 4.0;
Triple[][] normals = new Triple[steps + 1][steps + 1];
Triple sun = new Triple (3.6, 3.9, 0.6);

for (int i = 0; i < numTriangles; ++ i)
  for (int j = 0; j < 3; ++ j)
    normals[i][j] = new Triple (0.0, 0.0, 0.0);
/* compute triangle normals and vertex averaged normals */
for (int i = 0; i < numTriangles; ++ i) {
  Triple v0 = map[triangles[i].i[0]][triangles[i].j[0]],
    v1 = map[triangles[i].i[1]][triangles[i].j[1]],
    v2 = map[triangles[i].i[2]][triangles[i].j[2]];
  Triple normal = v0.subtract (v1).cross (v2.subtract (v1)).normalize
();
  triangles[i].n = normal;
  for (int j = 0; j < 3; ++ j) {
    normals[triangles[i].i[j]][triangles[i].j[j]] =
      normals[triangles[i].i[j]][triangles[i].j[j]].add (normal);
  }
}
/* compute vertex colors and triangle average colors */
for (int i = 0; i < numTriangles; ++ i) {
  RGB avg = new RGB (0.0, 0.0, 0.0);
  for (int j = 0; j < 3; ++ j) {
    int k = triangles[i].i[j], l = triangles[i].j[j];
    Triple vertex = map[k][l];
    RGB color = colors[k][l];
    Triple normal = normals[k][l].normalize ();
    Triple light = vertex.subtract (sun);
    double distance2 = light.length2 ();
    double dot = light.normalize ().dot (normal);
    double lighting = ambient + diffuse * ((dot < 0.0) ? - dot : 0.0)
/ distance2;
    color = color.scale (lighting);
    triangles[i].color[j] = color;
    avg = avg.add (color);
  }
  triangles[i].color = new Color (avg.scale (1.0 / 3.0).toRGB ());
}
```

一个明显的错误是这样做我们得不到尖利的边缘。例如，沿着阴影的边缘应该是不连续的边锋。但是按前面的算法，三角形两边的阴影将掩盖边缘。在地形中明显的颜色相交地带更为糟糕，比如蓝色的海洋和金色的沙滩。要解决这个问题，需要简单地在模型中再加入一些信息，并在一些顶点不共享颜色。

现在差不多已经完成所有在得到需要的地形数据结构之前的准备工作了。最后一件事，希望能注意到当我们计算地形中的部分是否面对太阳时，我们并不检查它是否被地形中的其他部分所遮挡。完成正确的阴影计算需要将突出的整体地形投影到下面的位置，就像从太阳上观测一样，然后再将三角形细分为阴影和无阴影部分，见图 7.54。

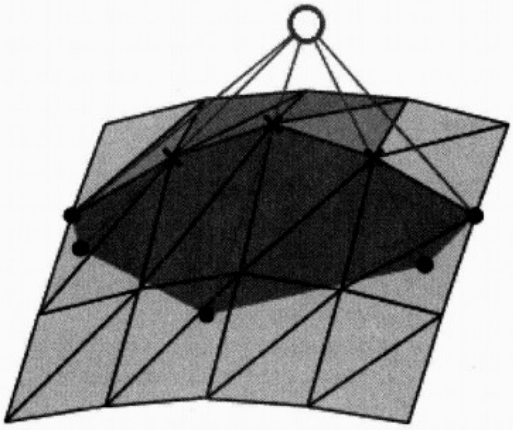


图 7.54　计算精确的阴影

但是这个过程的计算是缓慢而且困难的。代之的是使用一种简化的更为现实的权宜方法：对地形网格中的每一点，做它到太阳的连线，判断它是否与地形中的其他点相交，如果相交，则该点在阴影下。如果不相交，则计算光照。图 7.55 是带阴影的山地图。

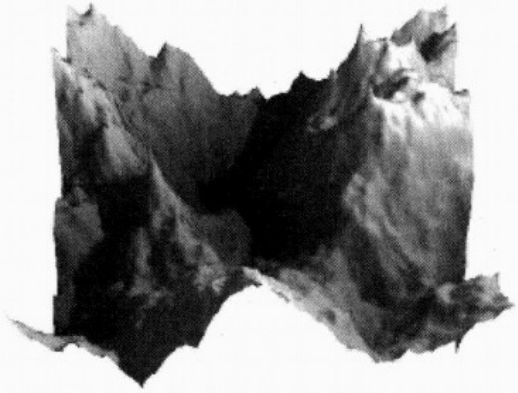


图 7.55　带阴影的山地图

更进一步，为了使这个计算更有效，我们并不做严格意义上的相交测试，代之的是仅取直线上的若干点，并检查地形上对应的点的高度是否高于直线，见图 7.56。

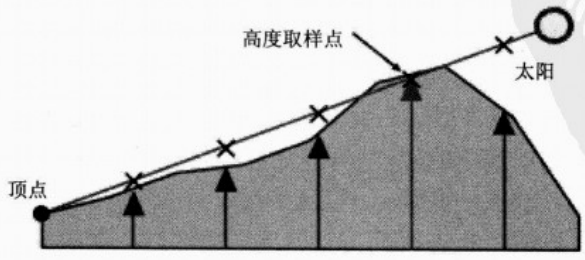


图 7.56　阴影的简单计算

对早先的阴影计算方法再稍做修改：如果一个顶点在阴影下，那么这里就没有散射

光线，只有周围的光。

```
double[][] shade = new double[steps + 1][steps + 1];
for (int i = 0; i <= steps; ++ i) {
  for (int j = 0; j <= steps; ++ j) {
    shade[i][j] = 1.0;
    Triple vertex = map[i][j];
    Triple ray = sun.subtract (vertex);
    double distance = steps * Math.sqrt (ray.x * ray.x + ray.z * ray.z);
    /* step along ray in horizontal units of grid width */
    for (double place = 1.0; place < distance; place += 1.0; {
      Triple sample = vertex.add (ray.scale (place / distance));
      double sx = sample.x, sy = sample.y, sz = sample.z;
      if ((sx < 0.0) || (sx > 1.0) || (sz < 0.0) || (sz > 1.0))
        break; /* steppd off terrain */
      double ground = exaggeration * terrain.getAltitude (sx, sz);
      if (ground >= sy) {
        shade[i][j] = 0.0;
        break;
      }
    }
  }
}
/* modified lighting computation:
    ...
    double shadow = shade[k][l];
    double lighting = ambient + diffuse * ((dot < 0.0) ? - dot : 0.0)
/
                                    distance2 * shadow;
*/
```

需要指出的是这个解决方法并不完美。如果在地形中有锋利的边缘或山顶，则会发现在阴影下的顶点有些可以计算出它在阴影下，有些却不能。一个典型的问题就是对于一些顶点的光线取样，如落在恰好处，于是我们得到阴影，而对其他点则只有得到光照。为了解决这个问题，我们需要一种适当的交集测试。事实上，只需使取阴影线上的样点更为密集就行。在这里，我们取线上的步长为网格的大小，如果发现错误的阴影，可以减小步长。

现在已经完成了一半的工作，拥有了一个由分形得到的着色、发光、渐变、附有阴影的地形。另外我们还有一个屏幕，现在要将两者结合起来。为了完成它，必须做好另外两样工作：第一，要计算出地形中每个三角投影在屏幕的什么地方，然后要画出这样的三角形。

7.4.4　观测点计算和变换世界

为了计算得到地形中三角应该绘制在屏幕的什么地方，我们需要知道观测者在虚拟空间中的位置和他观测的角度。

先看一下现实的世界：假设 Z 轴直指向你，沿着 Z 轴前进一米，想象一下你会看到什么？再退回你原来的位置，若你将现实世界沿 Z 轴减去一米，则你应该看到和刚才一

样的世界。事实上，你可以对你的任何移动在原位来做这样的想像：从 P 位置看到的世界，和从位置（0,0,0）看到的做 $-P$ 移动后的世界是相同的。

对观测方向可作同样的应用：从正向旋转 r 位置看到的世界和从不旋转看到的及旋转 $-r$ 后看到的世界是相同的。如果将移动和旋转结合，我们就得到：从旋转 r 角度的位置 P 看到的世界，和从原点正向看到的作过 $-P$ 平移和 $-r$ 旋转后的世界是一样的。

为什么要让人们如此地看世界？这是因为在原点正向绘制图形比在空间一个任意的位置要容易得多。所以，为了能让我们从任意位置观测，首先对空间做这样的变换，然后再实施简单的绘制过程。

我们先定义一些有关绘图的关键字：

1）现实空间。这是最初的空间，就是我们定义地形时对应的那个随意选取的坐标系空间。

2）视觉空间。这是我们已经看到的空间，也就是经过上述变换后的空间。如果你处于观测者的位置，对于现实空间作变换，就可以得到视觉空间。原点在观测点，Z 轴指向观测方向，Y 向上，X 向右。

3）屏幕空间。这是视觉空间转换到屏幕上的（x,y）象素坐标。

为了作现实空间和视觉空间的转换，我们需要表示出观测点和观测方向。观测点的表示很简单：$P_{you}=(X_{you},Y_{you},Z_{you})$。观测方向的表示也不麻烦。麻烦的是我们要从多种表现方法中选择一个来使用。

上面的表述见图 7.57。

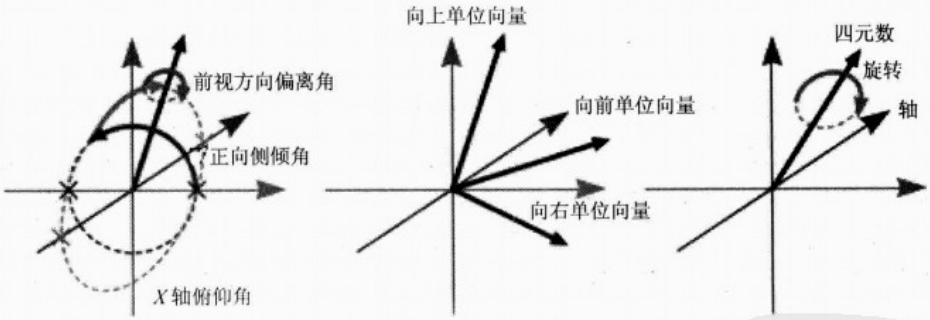


图 7.57　观察方向表示

遗憾的是，我们不能简单的用一个向量来代表人们的观测方向，因为这里并没有包含当前的直立方向的信息。

一种方法是用三种角度（Z 轴偏离角，X 轴俯仰角和正向侧倾角）来表示从你当前观察前视方向转换为所需的表示旋转的角度，但是，这样缺乏效率而且不易操作。另一种选择是使用一个 3×3 的旋转矩阵 M，这个矩阵本质上是由人们面向所指的单位向量，直立的单位向量和向右的单位向量组成。这是相当常用的表现方式，并且易于操作和理解，但是对于某些运算速度缓慢且甚受数字误差影响。

我们会代之以第三种方法：旋转四元数。正如第 6 章所述，这是一种有四个元素的数据结构（w, x, y, z）。(x, y, z)部分代表空间中的一个向量，w 在此表示这个向量的旋转量，

任何观测方向都可以用这种紧凑有效的结构表示。

7.4.5　空间旋转与四元数

四元数有多种用途，但是我们只考虑它在三维空间旋转的直接应用。下面的一些数学性质在第 6 章是曾经提到过的：

$$q=(w,x,y,z)$$

$$|q|^2=w^2+x^2+y^2+z^2$$

$$q^{-1}=(w/|q|^2,-x/|q|^2,-y/|q|^2,-z/|q|^2)$$

$$q_aq_b=(w_aw_b-x_ax_b-y_ay_b-z_az_b,w_ax_b+x_aw_b+y_az_b-z_ay_b,w_ay_b+y_aw_b+z_ax_b-x_az_b,$$
$$w_az_b+z_aw_b+x_ay_b-y_ax_b)$$

但是如何表示点的旋转呢？设有一向量 p 和一个旋转四元数 q_r，为了给这向量作旋转，我们将它转换成一个 w 系数为 0 的四元数，然后实施乘法再将其转换回来。下式中的符号 p^T 表示向量 p 的旋转。

$$p=(x,y,z)$$

$$q_p=(0,x,y,z)$$

$$q_p^T=q_rq_pq_r^{-1}=(w',x',y,'z')$$

$$p^T=(x',y',z')$$

我们用旋转四元数 q_r 乘以这个向量 p 转换成的四元数 q_p，再乘以旋转四元数的逆，结果就是旋转后的这个向量。这一点在第 6 章已经做了交代。

如果更注重速度，则还有一个选择：将旋转四元数转换为一个旋转矩阵，然后将一系列点用矩阵转换。但我们在这上面不作讨论。

一个旋转四元数是关于一个特殊向量或坐标轴的旋转。然而怎样计算去表示一个特殊旋转的四元数呢？假设有一个单位向量 v，我们需要一个四元数来表示对它作 w 的旋转，那么我们按下面的方法计算这个四元数。

向量：$v=(x,y,z);|v|=1.0$

旋转：w

旋转四元数：$(\cos(w),x\sin(w),y\sin(w),z\sin(w))$

如果我们想接着一个旋转再做下一个旋转，则可以明确地按顺序对向量做两次旋转，或者也可以简单地将这两个四元数相乘再将乘积作结合后的旋转操作。但是记住：$q_aq_b\neq q_bq_a$。

下面的代码实现了我们需要的基本四元数数学运算。

```
public class Quaternion {
  private double w, x, y, z;
  private Quaternion inverse;

  public Quaternion (double w, double x, double y, double z) {
    this.w = w;
    this.x = x;
    this.y = y;
    this.z = z;
```

```
    this.inverse = null;
  }

  public Quaternion inverse () {
    double scale = 1.0 / (x * x + y * y + z * z + w * w);
    return new Quaternion (w * scale, - x * scale, - y * scale, - z
* scale);
  }

  public Quaternion multiply (Quaternion q) {
    double qx = q.x, qy = q.y, qz = q.z, qw = q.w;
    double rw = w * qw - x * qx - y * qy - z * qz;
    double rx = w * qx + x * qw + y * qz - z * qy;
    double ry = w * qy + y * qw + z * qx - x * qz;
    double rz = w * qz + z * qw + x * qy - y * qx;
    return new Quaternion (rw, rx, ry, rz);
  }

  public Triple rotate (Triple t) {
    if (inverse == null)
      inverse = inverse ();

    double iw = inverse.w, ix = inverse.x, iy = inverse.y, iz =
inverse.z;
    double tx = t.x, ty = t.y, tz = t.z;

    double aw = - x * tx - y * ty - z * tz;
    double ax = w * tx + y * tz - z * ty;
    double ay = w * ty + z * tx - x * tz;
    double az = w * tz + x * ty - y * tx;

    double bx = aw * ix + ax * iw + ay * iz - az * iy;
    double by = aw * iy + ay * iw + az * ix - ax * iz;
    double bz = aw * iz + az * iw + ax * iy - ay * ix;

    return new Triple (bx, by, bz);
  }

  public static Quaternion newRotation (double r, double x, double y,
double z) {
    double len = Math.sqrt (x * x + y * y + z * z);
    double sin = Math.sin (r / 2.0);
    double cos = Math.cos (r / 2.0);
    double tmp = sin / len;
    return new Quaternion (cos, x * tmp, y * tmp, z * tmp);
  }
}
```

7.4.6　空间的转化

1. 现实空间化为视觉空间

需要将现实空间化为视觉空间的一些基本数学方法。下面代码就是用来实现这样的计算：

```
Triple loc = new Triple (0.5, 3.0, -2.0);
Quaternion rot = Quaternion.newRotation (-.82, 1.0, 0.0, 0.0);
Triple[][] eyeMap = new Triple[steps + 1][steps + 1];

for (int i = 0; i <= steps; ++ i) {
  for (int j = 0; j <= steps; ++ j) {
    Triple p = map[i][j];
    Triple t = p.subtract (loc);
    Triple r = rot.rotate (t);
    eyeMap[i][j] = r;
  }
}
```

2. 视觉空间化为屏幕空间

对空间进行的变换使得我们的视角变为从原点正视前方。现在需要将这个空间转换到屏幕上。若将屏幕看成空间中的一个窗口，则从眼中射出的光线将到达这个窗口上的每个像素并进入虚拟的屏幕，见图 7.58。

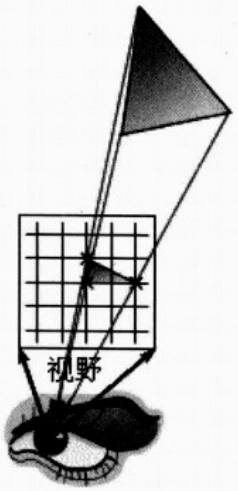


图 7.58　从视觉空间到屏幕空间

这样的计算是相当直观的。我们必须首先知道虚拟的屏幕与原点的距离（这个距离称为高度），如果有顶点落在这个平面的附近，我们就不对它进行变换，而是直接在窗口中画出我们看到的内容。否则，我们将坐标 x 和 y 除以 z，再按比例将结果绘制到屏幕上。

得到结果的比例大小取决于我们需要多宽的视野（fov, field of vision），如果我们要

确定水平方向的视野，则我们需要的大小为宽度除以 tan(*fov*/2)。常用的视野大小为 60 度。

```
double hither = .1;
double fov = Math.PI / 3;
XY[][] eyeMap = new XY[steps + 1][steps + 1];
int width = getSize ().width, height = getSize ().height;
double scale = width / 2 / Math.tan (fov / 2);

for (int i = 0; i <= steps; ++ i) {
  for (int j = 0; j <= steps; ++ j) {
    Triple p = eyeMap[i][j];
    double x = p.x, y = p.y, z = p.z;
    if (z >= hither) {
      double tmp = scale / z;
      scrMap[i][j] = new XY (width / 2 + (int) (x * tmp), height / 2
- (int) (y * tmp));
    } else {
      scrMap[i][j] = null;
    }
  }
}
```

现在我们有了每个地形顶点在屏幕空间的坐标，这就意味着我们有了每个三角形的顶点在屏幕上的象素坐标，接下来我们需要的就是将这些三角形画出来了。

7.4.7　屏幕上的绘制与收尾工作

最简单的绘制方法就是对屏幕上的每个三角形调用 Graphics.fillPolygon()，见图 7.52 和下面程序（paint）。但这并不十分有效。因为最后得到阴影平滑的三角形，使一些本应在地形中后部的三角却画到了前面。

```
public void paint (Graphics g) {
  for (int i = 0; i < numTriangles; ++ i) {
    XY xy0 = scrMap[triangles[i].i[0]][triangles[i].j[0]],
      xy1 = scrMap[triangles[i].i[1]][triangles[i].j[1]],
      xy2 = scrMap[triangles[i].i[2]][triangles[i].j[2]];
    double dot = - map[triangles[i].i[0]][triangles[i].j[0]]
      .subtract (loc).normalize ().dot (triangles[i].n);
    if ((dot > 0.0) && (xy0 != null) && (xy1 != null) && (xy2 != null))
{
      int[] x = { xy0.x, xy1.x, xy2.x }, y = { xy0.y, xy1.y, xy2.y };
      g.setColor (triangles[i].color);
      g.fillPolygon (x, y, 3);
    }
  }
}
```

人们可能设想只要对三角形进行排序并先画距离远的就行了。但是就像下面的图 7.59 所示，这种方法并非总有效。有些情形不能仅用简单的深度层次解决。图中的两个三角形有一个公共边，在下图的每个结构中，较小的三角形的第三个顶点（与 *Z* 轴相

比较）在公共边的后面，并在较大三角形的第三个顶点之前，所以一个深度排序不会改变它们的顺序。但是，在这个结构的左侧，较小的三角形在它同伴的前面，于是应该后画。在结构的右侧，它在前面的位置于是应该先画，见图 7.59。为了解决这个问题，我们需要一种更全面的算法。

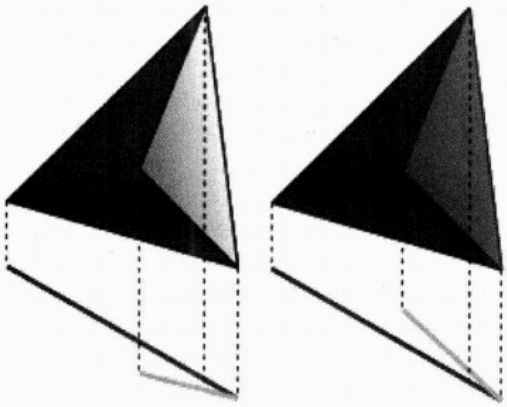


图 7.59　三角形的交叠

有多种方法可以解决这个问题。一种方法是建立跨度列表来描述屏幕上每条扫描线被三角形覆盖的部分，再用这个列表来解决重叠部分的问题。但是我们用另一种解决方案。当将三角形画在一张缓冲的图像中，除了存储每个像素的红、绿、蓝系数，还存储屏幕上每个像素的深度。当我们在屏幕上绘制像素之前，应先参考缓冲值来了解是否已经绘制过相邻的点。如果已经绘制过，则不画这个像素，否则，将颜色和深度信息照常存储。用这种方法，就解决了任意三角形在像素精度的交叠问题，见图 7.60。

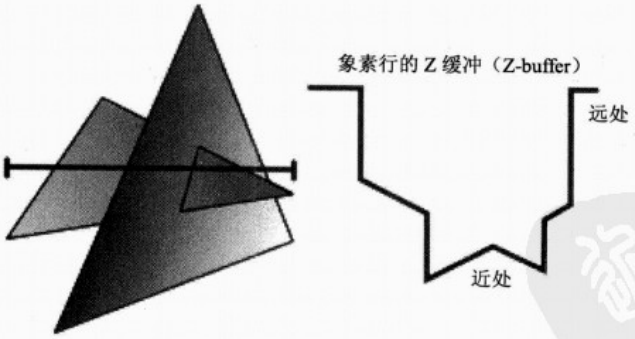


图 7.60　Z-缓冲器

现在已知三角形在屏幕的坐标和解决多边形重叠问题的算法，要做的只是将三角形画出来就行了。对每个三角形我们将用简单的算法来将三角形的顶点垂直地排序，再从上至下对每条扫描线计算三角形的左边和右边并用 setPixel()函数绘制中间的跨度部分。像以前一样，将三角形绘制在一个 32 位的缓冲内存中并用 java.awt.Image 类来显示它。

```
import java.awt.*;
import java.awt.image.*;
```

```
import java.io.*;

public class Rasteriser {
  public int width, height;
  private int[] buffer;
  private double[] zBuffer;
  private MemoryImageSource source;
  private Image image;

  public Rasteriser (int width, int height) {
    this.width = width;
    this.height = height;
    buffer = new int[width * height];
    zBuffer = new double[width * height];
    source = new MemoryImageSource (width, height, buffer, 0, width);
    source.setAnimated (true);
    clear ();
  }

  public Image createImage (Component component) {
    return component.createImage (source);
  }

  public void clear () {
    for (int i = 0; i < width * height; ++ i) {
      buffer[i] = 0xff000000; // black
      zBuffer[i] = Double.MAX_VALUE; // distant
    }
  }

  // rasterize an arbitrary triangle
  public void rasterise (Vertex a, Vertex b, Vertex c) {
    // sort vertices vertically
    if (b.xy.y < a.xy.y) {
      Vertex tmp = b;
      b = a;
      a = tmp;
    }
    if (c.xy.y < a.xy.y) {
      Vertex tmp = c;
      c = a;
      a = tmp;
    }
    if (c.xy.y < b.xy.y) {
      Vertex tmp = c;
      c = b;
      b = tmp;
    }
    if (a.xy.y == b.xy.y) { // flat top
      if (b.xy.x < a.xy.x) { // sort horizontally
```

```
    Vertex tmp = b;
    b = a;
    a = tmp;
   }
   double dx = (b.xy.x == a.xy.x) ? 1.0 : 1.0 / (b.xy.x - a.xy.x);
   double dy = (c.xy.y == a.xy.y) ? 1.0 : 1.0 / (c.xy.y - a.xy.y);
   subRasterise (a.xy.y, c.xy.y,
               a.xy.x, (c.xy.x - a.xy.x) * dy,
               b.xy.x, (c.xy.x - b.xy.x) * dy,
               a.z, (b.z - a.z) * dx, (c.z - a.z) * dy,
               a.rgb, b.rgb.subtract (a.rgb).scale (dx),
               c.rgb.subtract (a.rgb).scale (dy));
  } else if (b.xy.y == c.xy.y) { // flat bottom
   if (c.xy.x < b.xy.x) { // sort horizontally
    Vertex tmp = c;
    c = b;
    b = tmp;
   }
   double dx = (c.xy.x == b.xy.x) ? 1.0 : 1.0 / (c.xy.x - b.xy.x);
   double dy = (b.xy.y == a.xy.y) ? 1.0 : 1.0 / (b.xy.y - a.xy.y);
   subRasterise (a.xy.y, b.xy.y,
               a.xy.x, (b.xy.x - a.xy.x) * dy,
               a.xy.x, (c.xy.x - a.xy.x) * dy,
               a.z, (c.z - b.z) * dx, (b.z - a.z) * dy,
               a.rgb, c.rgb.subtract (b.rgb).scale (dx),
               b.rgb.subtract (a.rgb).scale (dy));
  } else { // must be split
   double mid = (double) (b.xy.y - a.xy.y) / (c.xy.y - a.xy.y);
   double midX = a.xy.x + mid * (c.xy.x - a.xy.x);
   double midZ = a.z + mid * (c.z - a.z);
   RGB midRGB = a.rgb.add (c.rgb.subtract (a.rgb).scale (mid));
   if (b.xy.x < midX) { // long edge on right
    double dx = (midX == b.xy.x) ? 1.0 : 1.0 / (midX - b.xy.x);
    double dy = 1.0 / (b.xy.y - a.xy.y);
    subRasterise (a.xy.y, b.xy.y,
                a.xy.x, (b.xy.x - a.xy.x) * dy,
                a.xy.x, (midX - a.xy.x) * dy,
                a.z, (midZ - b.z) * dx, (b.z - a.z) * dy,
                a.rgb, midRGB.subtract (b.rgb).scale (dx),
                b.rgb.subtract (a.rgb).scale (dy));
    dy = 1.0 / (c.xy.y - b.xy.y);
    subRasterise (b.xy.y, c.xy.y,
                b.xy.x, (c.xy.x - b.xy.x) * dy,
                midX, (c.xy.x - midX) * dy,
                b.z, (midZ - b.z) * dx, (c.z - b.z) * dy,
                b.rgb, midRGB.subtract (b.rgb).scale (dx),
                c.rgb.subtract (b.rgb).scale (dy));
   } else { // long edge on left
    double dx = (b.xy.x == midX) ? 1.0 : 1.0 / (b.xy.x - midX);
    double dy = 1.0 / (b.xy.y - a.xy.y);
```

```
      subRasterise (a.xy.y, b.xy.y,
                    a.xy.x, (midX - a.xy.x) * dy,
                    a.xy.x, (b.xy.x - a.xy.x) * dy,
                    a.z, (b.z - midZ) * dx, (midZ - a.z) * dy,
                    a.rgb, b.rgb.subtract (midRGB).scale (dx),
                    midRGB.subtract (a.rgb).scale (dy));
      dy = 1.0 / (c.xy.y - b.xy.y);
      subRasterise (b.xy.y, c.xy.y,
                    midX, (c.xy.x - midX) * dy,
                    b.xy.x, (c.xy.x - b.xy.x) * dy,
                    midZ, (b.z - midZ) * dx, (c.z - midZ) * dy,
                    midRGB, b.rgb.subtract (midRGB).scale (dx),
                    c.rgb.subtract (midRGB).scale (dy));
    }
  }
}

// rasterize a flat triangle
public void subRasterise (int y0, int y1, double x0, double dX0DY,
double x1, double dX1DY, double z, double dZDX, double dZDY, RGB color,
RGB dColorDX, RGB dColorDY) {
  for (int j = y0; j <= y1; ++ j) {
    int i0 = (int) x0, i1 = (int) x1;
    RGB tmpColor = color;
    double tmpZ = z;
    for (int i = i0; i <= i1; ++ i) {
      setPixel (i, j, tmpZ, tmpColor);
      tmpZ += dZDX;
      tmpColor = tmpColor.add (dColorDX);
    }
    x0 += dX0DY;
    x1 += dX1DY;
    z += dZDY;
    color = color.add (dColorDY);
  }
}

// set a pixel if zbuffer says okay
public final void setPixel (int x, int y, double z, RGB rgb) {
  if ((x >= 0) && (x < width) && (y >= 0) && (y < height)) {
    int o = x + y * width;
    if (z < zBuffer[o]) {
      zBuffer[o] = z;
      buffer[o] = rgb.toRGB ();
    }
  }
}

public void update () {
  source.newPixels ();
```

```
    }

    public void savePPM (String file) throws IOException {
      FileOutputStream fileOut = new FileOutputStream (file);
      BufferedOutputStream bufferedOut = new BufferedOutputStream (fileOut);
      DataOutputStream dataOut = new DataOutputStream (bufferedOut);
      dataOut.write ("P6\n".getBytes ());
      dataOut.write ((width + " " + height + "\n").getBytes ());
      dataOut.write ("255\n".getBytes ());
      for (int i = 0; i < width * height; ++ i) {
        int rgb = buffer[i];
        dataOut.write (rgb >> 16);
        dataOut.write (rgb >> 8);
        dataOut.write (rgb >> 0);
      }
      dataOut.close ();
    }
  }
```

到此为止，我们构建了地形并给它光照、渐变和阴影，再按它们的位置和视角进行变换，然后将得到的三角形绘制到屏幕上。其实，还有很多办法使得图片更有趣。例如，对图 7.61，图中有深蓝色的天空作为背景和一个分形的星球。我们如何来画得此图？

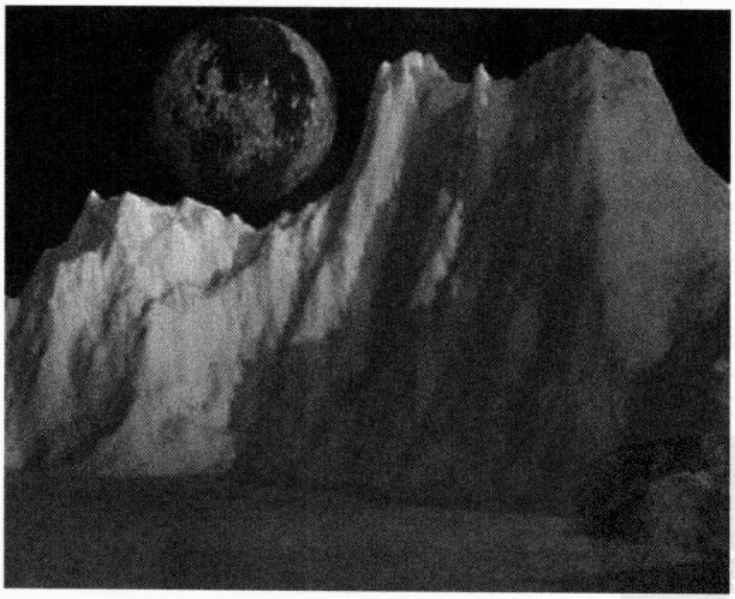


图 7.61　山地和星空

首先，我们修改了 rasterizer 类来预先载入一张背景图片取代以往的黑色背景。然后，再生成一张从蓝色到黑色渐变的分形地形，再修改显示部分的代码让它从上观测。很快地，一幅深蓝色天空（不是真实的）背景的图形就得到了，再稍作修改就可以得到云（见图 7.62）。如果希望云在透视时会折射，就像在星球的地平线上那样，我们只需定义一个合适的地形函数就行。

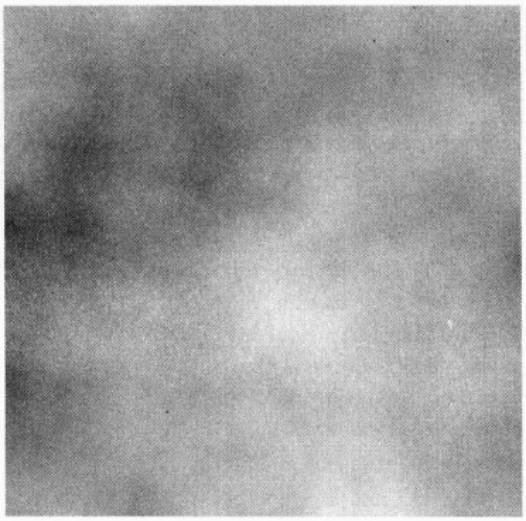


图 7.62　分形云

接下来加入了海洋。为此使用产生分形的方法并修改 getAltitude()方法来得到一个固定的切面，于是所有低于这个高度的地方都化为一个固定的海岸线高度（蓝色的），然后再用一个新的分形地形来得到一个星球的表面，见图 7.63。

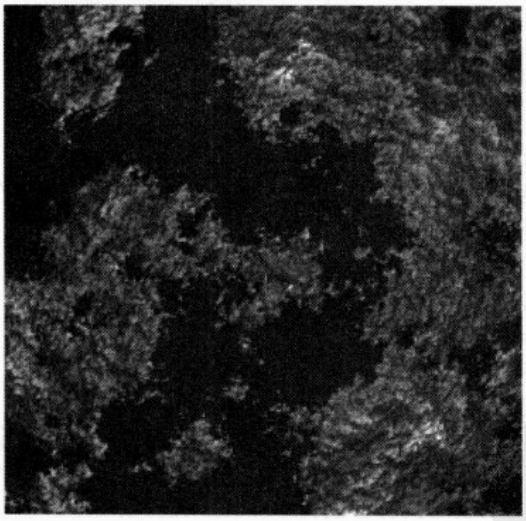


图 7.63　俯看山地和海洋

将这个分形地形图像作为纹理粘上，就得到了一个星球。最后，将太空作为背景和星球结合，在前方画上一个粉红色的地形，图片就完成了。当然，还可以加入另一个光源来使阴影更加柔和。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Landscape extends Applet {
```

```
private static final int lod = 5, steps = 1 << lod;
private static final double roughness = 0.5;
private static final double exaggeration = 0.7;
private static final double ambient = 0.3;
private static final double diffuse = 0.7;
private static final double strength = 8.0;

static final double hither = .1;
static final double fov = Math.PI / 6;

private Terrain terrain;
private int width, height;

private Triple[][] map;
private RGB[][] colors;
private double[][] shade;
private Triple[][] normals;
private Triple[][] eyeMap;
private XY[][] scrMap;

private int numTriangles = steps * steps * 2;
private Triangle[] triangles;

private Triple sun = new Triple (3.6, 1.5, 0.6);
private Triple loc = new Triple (0.4, 2.4, -1.6);
private Quaternion rot = Quaternion.newRotation (-.82, 1.0, 0.0, 0.0);

private Rasteriser rasteriser;

public void init () {
  setBackground (Color.black);

  map = new Triple[steps + 1][steps + 1];
  colors = new RGB[steps + 1][steps + 1];
  shade = new double[steps + 1][steps + 1];
  normals = new Triple[steps + 1][steps + 1];
  eyeMap = new Triple[steps + 1][steps + 1];
  scrMap = new XY[steps + 1][steps + 1];
  triangles = new Triangle[numTriangles];

  terrain = new FractalTerrain (lod, roughness);

  // fill in vertices
  for (int i = 0; i <= steps; ++ i) {
    for (int j = 0; j <= steps; ++ j) {
      double x = 1.0 * i / steps, z = 1.0 * j / steps;
      double altitude = exaggeration * terrain.getAltitude (x, z);
      map[i][j] = new Triple (x, altitude, z);
      colors[i][j] = terrain.getColor (x, z);
      shade[i][j] = 1.0;
```

```
    normals[i][j] = new Triple (0.0, 0.0, 0.0);
  }
}

// set up tessellation
int triangle = 0;
for (int i = 0; i < steps; ++ i) {
  for (int j = 0; j < steps; ++ j) {
    triangles[triangle ++] = new Triangle (i, j, i + 1, j, i, j + 1);
    triangles[triangle ++] = new Triangle (i + 1, j, i + 1, j + 1, i, j + 1);
  }
}

// compute triangle and vertex normals
for (int i = 0; i < numTriangles; ++ i) {
  Triple v0 = map[triangles[i].i[0]][triangles[i].j[0]],
    v1 = map[triangles[i].i[1]][triangles[i].j[1]],
    v2 = map[triangles[i].i[2]][triangles[i].j[2]];
  Triple normal = v0.subtract (v1).cross (v2.subtract (v1)).normalise ();
  triangles[i].n = normal;
  for (int j = 0; j < 3; ++ j) {
    normals[triangles[i].i[j]][triangles[i].j[j]] =
      normals[triangles[i].i[j]][triangles[i].j[j]].add (normal);
  }
}

// compute vertex shadows
for (int i = 0; i <= steps; ++ i) {
  for (int j = 0; j <= steps; ++ j) {
    shade[i][j] = 1.0;
    Triple vertex = map[i][j];
    Triple ray = sun.subtract (vertex);
    double distance = steps * Math.sqrt (ray.x * ray.x + ray.z * ray.z);
    /* step along ray in horizontal units of grid width */
    for (double place = 1.0; place < distance; place += 1.0) {
      Triple sample = vertex.add (ray.scale (place / distance));
      double sx = sample.x, sy = sample.y, sz = sample.z;
      if ((sx < 0.0) || (sx > 1.0) || (sz < 0.0) || (sz > 1.0))
        break; /* stepped off terrain */
      double ground = exaggeration * terrain.getAltitude (sx, sz);
      if (ground >= sy) {
        shade[i][j] = 0.0;
        break;
      }
    }
  }
}

// compute triangle and vertex colors
for (int i = 0; i < numTriangles; ++ i) {
```

```
    RGB average = new RGB (0.0, 0.0, 0.0);
    for (int j = 0; j < 3; ++ j) {
      int k = triangles[i].i[j], l = triangles[i].j[j];
      Triple vertex = map[k][l];
      RGB color = colors[k][l];
      double shadow = shade[k][l];
      Triple normal = normals[k][l].normalise ();
      Triple light = vertex.subtract (sun);
      double distance2 = light.length2 ();
      double dot = light.normalise ().dot (normal);
      double lighting = ambient;
      if (dot < 0.0) // add diffuse lighting
        lighting -= strength * diffuse * dot * shadow / distance2;
      color = color.scale (lighting);
      triangles[i].rgb[j] = color;
      average = average.add (color);
    }
    triangles[i].color = new Color (average.scale (1.0 / 3.0).toRGB ());
  }

  width = getSize ().width / 2;
  height = getSize ().height;

  worldToEye ();
  eyeToScreen ();

  // only initialize AWT once
  if (rasteriser == null) {
    rasteriser = new Rasteriser (width, height);
    Viewer viewer = new Viewer (rasteriser);
    setLayout (new FlowLayout (FlowLayout.RIGHT, 0, 0));
    add (viewer);
  }
  enableEvents (AWTEvent.KEY_EVENT_MASK | AWTEvent.MOUSE_EVENT_MASK);
  requestFocus ();
}

protected void processMouseEvent (MouseEvent ev) {
  if (ev.getID () == MouseEvent.MOUSE_PRESSED)
    requestFocus ();
  super.processMouseEvent (ev);
}

protected void processKeyEvent (KeyEvent ev) {
  if (ev.getID () != KeyEvent.KEY_PRESSED) {
    super.processKeyEvent (ev);
    return;
  }
  int key = ev.getKeyCode ();
  // rotate
```

```
      if ((key == KeyEvent.VK_UP) || (key == KeyEvent.VK_DOWN) ||
          (key == KeyEvent.VK_LEFT) || (key == KeyEvent.VK_RIGHT) ||
          (key == KeyEvent.VK_COMMA) || (key == KeyEvent.VK_PERIOD)) {
        double r = ((key == KeyEvent.VK_DOWN) || (key == KeyEvent.VK_RIGHT) ||
                    (key == KeyEvent.VK_PERIOD)) ? - Math.PI / 64 : Math.PI / 64;
        double x = ((key == KeyEvent.VK_UP)||(key == KeyEvent.VK_DOWN))?1.0 : 0.0,
        y = ((key == KeyEvent.VK_LEFT) || (key == KeyEvent.VK_RIGHT)) ? 1.0 : 0.0,
        z = ((key == KeyEvent.VK_COMMA)||(key == KeyEvent.VK_PERIOD))?1.0: 0.0;
        Quaternion delta = Quaternion.newRotation (r, x, y, z);

        rot = delta.multiply (rot);

        worldToEye ();
        eyeToScreen ();
        repaint ();
      // translate
      } else if ((key == KeyEvent.VK_Z) || (key == KeyEvent.VK_X)) {
        double z = (key == KeyEvent.VK_Z) ? 1.0 / 8: -1.0 / 8;
        Triple delta = new Triple (0.0, 0.0, z);
        Quaternion tmp = rot.inverse ();
        Triple deltar = tmp.rotate (delta);
        loc = loc.add (deltar);

        worldToEye ();
        eyeToScreen ();
        repaint ();
      // rasterize
      } else if (key == KeyEvent.VK_ENTER) {
        rasteriser.clear ();
        for (int i = 0; i < numTriangles; ++ i) {
          XY xy0 = scrMap[triangles[i].i[0]][triangles[i].j[0]],
            xy1 = scrMap[triangles[i].i[1]][triangles[i].j[1]],
            xy2 = scrMap[triangles[i].i[2]][triangles[i].j[2]];
          double dot = - map[triangles[i].i[0]][triangles[i].j[0]].
subtract (loc).normalise ().dot (triangles[i].n);
          dot = 1.0;
          if ((dot > 0.0) && (xy0 != null) && (xy1 != null) && (xy2 != null)) {
            double z0 = eyeMap[triangles[i].i[0]][triangles[i].j[0]].z,
              z1 = eyeMap[triangles[i].i[1]][triangles[i].j[1]].z,
              z2 = eyeMap[triangles[i].i[2]][triangles[i].j[2]].z;
            Vertex v0 = new Vertex (xy0, z0, triangles[i].rgb[0]),
              v1 = new Vertex (xy1, z1, triangles[i].rgb[1]),
              v2 = new Vertex (xy2, z2, triangles[i].rgb[2]);
            rasteriser.rasterise (v0, v1, v2);
          }
        }
        rasteriser.update ();
      // new landscape
      } else if (key == KeyEvent.VK_SPACE) {
        init ();
```

```
    repaint (0, 0, width, height);
  }
  super.processKeyEvent (ev);
}

void worldToEye () {
  for (int i = 0; i <= steps; ++ i) {
    for (int j = 0; j <= steps; ++ j) {
      Triple p = map[i][j];
      Triple t = p.subtract (loc);
      Triple r = rot.rotate (t);
      eyeMap[i][j] = r;
    }
  }
}

void eyeToScreen () {
  double scale = width / 2 / Math.tan (fov / 2);
  for (int i = 0; i <= steps; ++ i) {
    for (int j = 0; j <= steps; ++ j) {
      Triple p = eyeMap[i][j];
      double x = p.x, y = p.y, z = p.z;
      if (z >= hither) {
        double tmp = scale / z;
        scrMap[i][j] = new XY (width / 2 + (int) (x * tmp), height
        / 2 - (int) (y * tmp));
      } else {
        scrMap[i][j] = null;
      }
    }
  }
}

public void paint (Graphics g) {
  for (int i = 0; i < numTriangles; ++ i) {
    XY xy0 = scrMap[triangles[i].i[0]][triangles[i].j[0]],
      xy1 = scrMap[triangles[i].i[1]][triangles[i].j[1]],
      xy2 = scrMap[triangles[i].i[2]][triangles[i].j[2]];
    double dot = - map[triangles[i].i[0]][triangles[i].j[0]]. subtract
    (loc).normalise ().dot (triangles[i].n);
    if ((dot > 0.0) && (xy0 != null) && (xy1 != null) && (xy2 != null)) {
      int[] x = { xy0.x, xy1.x, xy2.x }, y = { xy0.y, xy1.y, xy2.y };
      g.setColor (triangles[i].color);
      g.fillPolygon (x, y, 3);
    }
  }
}
public Dimension getPreferredSize () {
  return new Dimension (512, 256);
}
```

```
    public static void main (String[] args) {
      Frame frame = new Frame ("Landscape");
      Landscape landscape = new Landscape ();
      frame.add (landscape, "Center");
      frame.pack ();
      landscape.init ();
      frame.setVisible (true);
    }
  }
```

第 8 章　奇异吸引子

分形与混沌动力系统有紧密的关系。当系统的混沌运动在相空间（一个有 n 个一阶常微分方程组的系统，它的 $2n$ 个元素 $\{(x_i,\dot{x}_i):i=1,2,\cdots,n\}$ 组成的 $2n$ 维空间称为相空间，当 $n=2$ 时称相平面）中无穷地缠绕、折叠和扭结，构成了具有无穷层次的自相似结构时，这种结构称为奇异吸引子。不仅在相空间，由若干个非线性代数方程组组成的系统，其解在空间的运动轨迹，也有可能构成了具有无穷层次的自相似结构，它们的几何维数是分维的，它们也是奇异吸引子，也是分形。

本章将主要介绍这两类分形，以及由其他变换形成的奇异吸引子分形。

8.1　洛伦兹吸引子

1961 年，美国气象学家洛伦兹（Lorenz）在用计算机进行天气预报计算时，为了考察一个很长的序列，他没令计算机从头算起而令计算机中途开始，把一个上次的输出值直接作为初始值输入计算机，然后他下楼去客厅喝咖啡。一小时后他上来看到的结果却使他大为惊异，原来计算机显示的结果与上次的模式大有偏离。进一步的计算表明，输入的细微区别可能很快引起输出的巨大差别。1979 年，洛伦兹在一次演讲中的题目是："可预言性：一只蝴蝶在巴西扇动了一下翅膀，会引起得克萨斯的龙卷风吗？"这就是著名的"蝴蝶效应"——对初始条件的敏感依赖性。下面几个偏微分方程组就是他建立的。

8.1.1　洛伦兹 I 型

这是洛伦兹发现的第一个产生混沌现象的也是其最著名的非线性微分方程组

$$\left.\begin{aligned}\dot{x}&=a(y-x)\\ \dot{y}&=x(b-z)-y\\ \dot{z}&=xy-cz\end{aligned}\right\}\tag{8.1}$$

式中 $\dot{x}$（$\dot{y},\dot{z}$）表示 x（y,z）对时间 t 求导。洛伦兹采用常数 $a=10,b=28,c=8/3$，另一组是 $a=28,b=46.92,c=4$。有时称 a 为 Prandtl 数，b 为 Rayleigh（雷利）数。系统（8.1）既不能形成极限环（一个吸引集，它的轨道或轨线收敛且轨线具有周期性），也不能达到一个稳定状态，代之的是一个确定性的混沌。简言之，混沌与分形有着密切关系，混沌运动的轨道或奇异吸引子都是分形。混沌运动是高度无序的也是混乱的（不是简单的混乱），这种性质都被反映在分形的无穷复杂性上面。分形学是描述混沌现象的恰当语言，分形的维数成了研究混沌现象的一个定量参数。像其他混沌系统一样，洛伦兹系统对初值很敏感。不管两个初始状态如何地挨近，它终将还是离散。尽管方程组看起来是足够地简单，但它还是引出了令人惊异的轨道，即奇异吸引子。

当 $a=3$，$b=26.5$，$c=1$ 时，得到图 8.1。图 8.2 是换一个角度看其图形。

1963 年，洛伦兹把他的发现发表在《大气科学》杂志（Journal of the Atmospheric Sciences）上。但直到下一个 10 年内，它却并未引起人们的注意。他的另一篇文章“确定性的非周期流”（deterministic nonperiodic flow）叙述了现在我们所称的“混沌”。

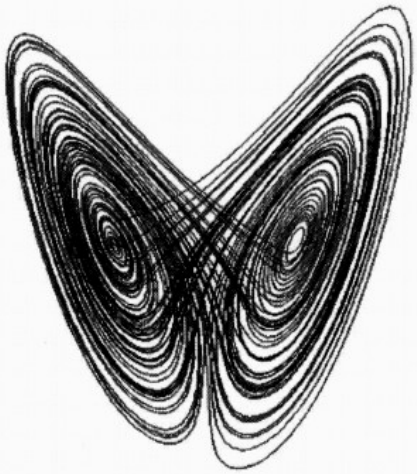


图 8.1　洛伦兹奇异吸引子之一

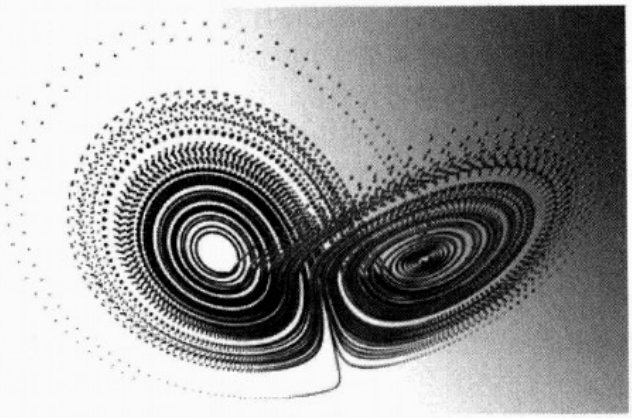


图 8.2　洛伦兹奇异吸引子之二

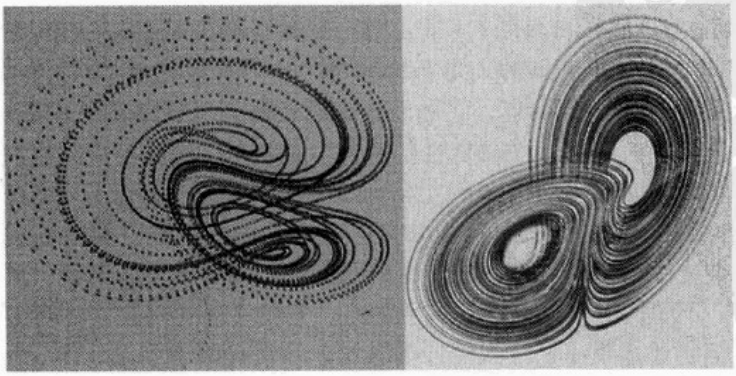


图 8.3　洛伦兹奇异吸引子之三

图 8.3 是该方程组得到的另两张图。

产生洛伦兹奇异吸引子的 C 程序见程序 8.1。根据迭代公式，该程序求出吸引子图形中每个点的三维坐标。如果想在屏幕上绘制吸引子图形，只需把 printf 命令改成画点或画线命令即可，请参照程序 8.2。

程序 8.1　洛伦兹奇异吸引子之一。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 10000

int main(int argc,char **argv)
{
   int i=0;
   double x0,y0,z0,x1,y1,z1;
   double h = 0.01;
   double a = 10.0;
   double b = 28.0;
   double c = 8.0 / 3.0;

   x0 = 0.1;
   y0 = 0;
   z0 = 0;
   for (i=0;i<N;i++) {
      x1 = x0 + h * a * (y0 - x0);//h 是计算时的步长,
      y1 = y0 + h * (x0 * (b - z0) - y0);//这里的迭代就是公
      z1 = z0 + h * (x0 * y0 - c * z0);//式(8.1)
      x0 = x1;
      y0 = y1;
      z0 = z1;
      if (i > 100)
        printf("%d %g %g %g\n",i,x0,y0,z0);
   }
}
```

产生图 8.4 且随时间连续动画的 Java Applet 程序见程序 8.2。

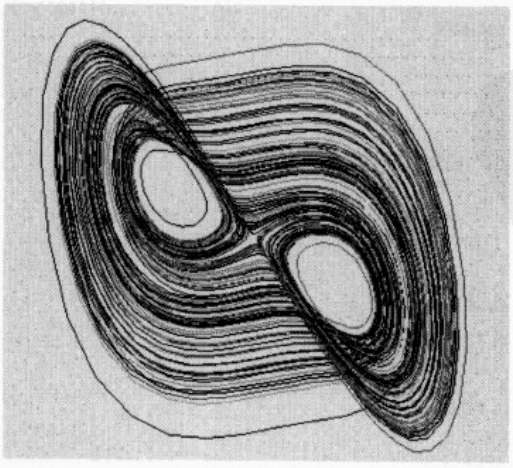


图 8.4　洛伦兹奇异吸引子之四

程序 8.2　洛伦兹奇异吸引子之二。

```
// Lorenz's strange attractor

import java.awt.*;

public class Attract extends java.applet.Applet implements Runnable {
  Thread attract=null;
  double x, y, z;

  public void init() {
    setBackground(Color.black);
    setForeground(Color.white);
    x=0; y=20; z=20;
  }

  public void paint(Graphics g) {
    double xx, yy, zz;

    xx=x+0.01*(-10*x+10*y);
    yy=y+0.01*(28*x-y-x*z);
    zz=z+0.01*(-8*z/3+x*y);
    g.drawLine((int)(x*10+250),(int)(-z*10+500),(int)(xx*10+250),
               (int)(-zz*10+500));
    x=xx; y=yy; z=zz;
  }

/*
 public void update(Graphics g) {
       g.setColor(getBackground());
       g.fillRect(0, 0, width, height);
       g.setColor(getForeground());
       paint(g);
    }
*/

  public void update(Graphics g) {
    paint(g);
  }

  public void start() {
    if (attract==null) {
      attract = new Thread(this);
      attract.start();
    }
  }
```

```
    public void stop() {
      if (attract!=null) {
        attract.stop();
        attract=null;
      }
    }

    public void run() {
      while (true) {
        try {
          Thread.currentThread().sleep(10);
        } catch (InterruptedException e) {
          break;
        }
        repaint();
      }
    }
  }
```

8.1.2　洛伦兹 II 型

这个非线性微分方程组是洛伦兹为演示动力系统的混沌行为而提出的。

$$\left.\begin{aligned}\dot{x}&=x+ad(y-x)\\ \dot{y}&=y+d(bx-y-zx)\\ \dot{z}&=z+d(xy-cz)\end{aligned}\right\}\tag{8.2}$$

它有四个参数 a,b,c 和 d，当 $d>0.3$ 时会产生许多有趣的奇异吸引子，而这些奇异吸引子就是分形。图 8.5 是当 $a=2.01342$，$b=6.19181$，$c=0.85914$ 和 $d=0.14430$ 时，解上述方程组画出的图形。当 $a=0.632911$，$b=6.202531$，$c=2.405063$ 和 $d=0.357594$ 时得到图 8.6。

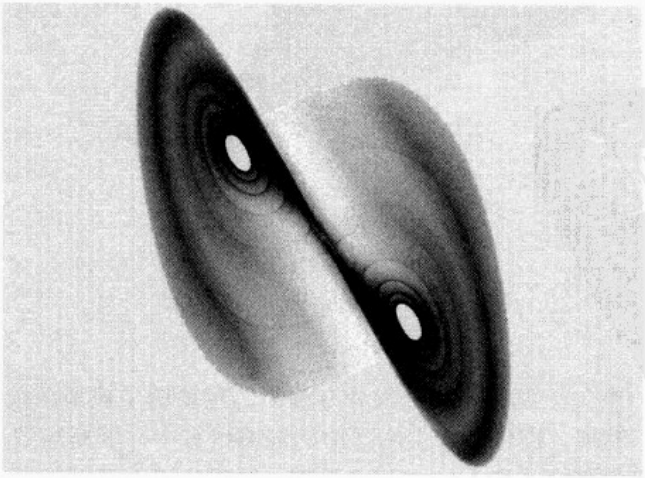


图 8.5　洛伦兹奇异吸引子之五

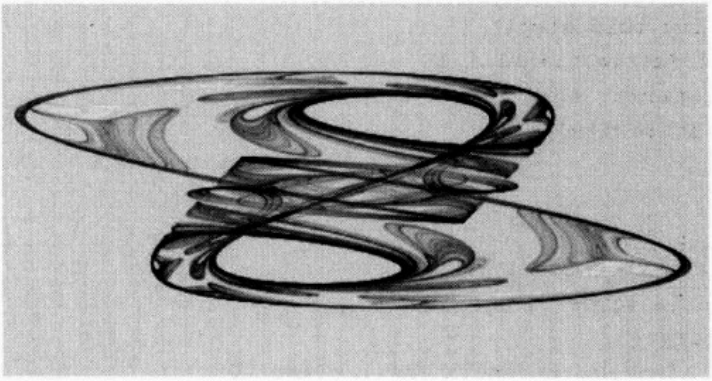


图 8.6　洛伦兹奇异吸引子之六

8.1.3　洛伦兹 84

这个方程组首先出现在洛伦兹的论文“无规律性:大气层的基本性质”(Irregularity: a fundamental property of the atmosphere）上，该方程组是长期大气循环的低维模型。它不是现时大气事件的图形表示，其轨道坐标是三维的。

$$\left.\begin{aligned}\dot{x} &= x + d(-ax - y^2 - z^2 + af)\\ \dot{y} &= y + d(-y + xy - bxz + g)\\ \dot{z} &= z + d(-z + bxy + xz)\end{aligned}\right\} \tag{8.3}$$

当参数 $a=1.111$，$b=1.479$，$d=4.494$，$g=0.44$，$f=0.135$ 时，可画得图 8.7。

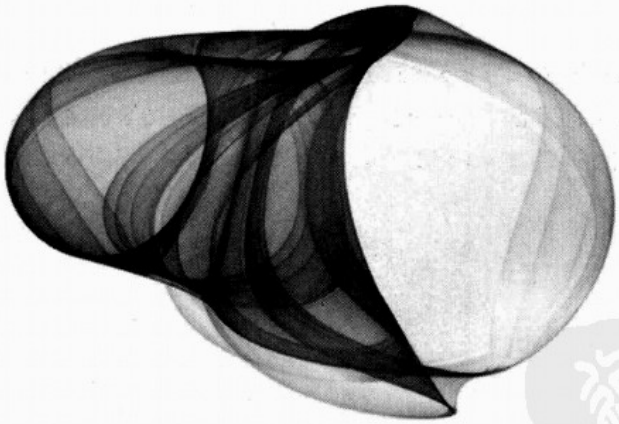


图 8.7　洛伦兹奇异吸引子之七

8.2　罗斯洛吸引子

1976 年，另一个关于混沌非线性微分方程组的重要例子被德国医学博士奥特。罗斯洛（Otto Rossler）发现。在化学和生物学中的混沌研究中，罗斯洛致力于发现比洛伦兹系统更为简单的且展现混沌行为的方程组系统，这就是现在著名的罗斯洛方程组。虽然它不如洛伦兹奇异吸引子来得出名，但其图形更为奇特。该方程组如下：

$$\left.\begin{aligned}\dot{x} &= -y - z \\ \dot{y} &= x + ay \\ \dot{z} &= b + z(x - c)\end{aligned}\right\} \tag{8.4}$$

其中 $a=0.2$, $b=0.2$, $c=5.7$。系统（8.4）也不能形成极限环更不能达到一个稳定状态，得到的只是一个确定性的混沌。也像其他混沌系统一样，罗斯洛系统对初值非常敏感。不管两个初始状态如何地接近，最终还是发散。尽管方程组看起来并不复杂，但它还是产生出令人眼花缭乱和奇异的轨道，即奇异吸引子。几个例子如图 8.8～图 8.10 所示。产生图形的程序见程序 8.3，需说明的是，这个程序用到了 OpenGL 库。

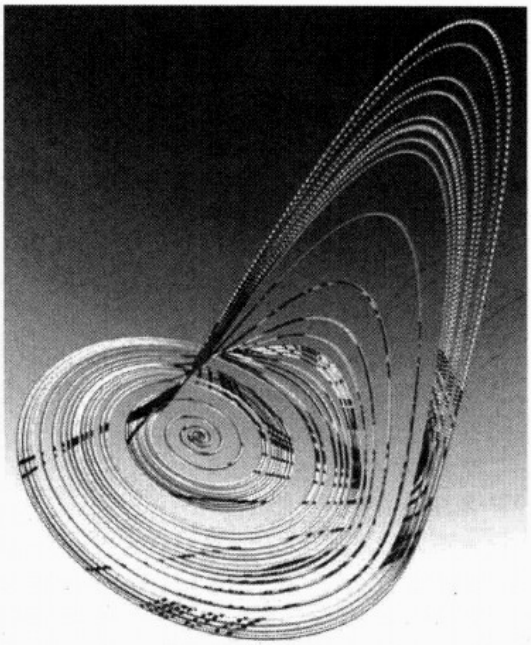


图 8.8　罗斯洛吸引子之一

程序 8.3　罗斯洛吸引子。

```
#include <GL/glut.h>
#include <stdlib.h>

GLfloat diffuseMaterial[4] = {0.5, 0.5, 0.5, 1.0};

void init(void)
{
   GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glEnable(GL_DEPTH_TEST);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

```
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);

    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}

void display(void)
{
    GLfloat x0, y0, z0, x1, y1, z1;
    GLfloat h;
    int Puntos, MaxPuntos=3850;

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glLoadIdentity ();
    gluLookAt (0.0, -20.0, 37.0, 0.0, 12.0, 0.0, 0.0, 1.0, 0.0);
    glColor3f(0.2f, 0.8f, 0.8f);
    x0=1.0f;
    y0=1.0f;
    z0=1.0f;
    Puntos=0;
    h=0.02f;
    while  (Puntos<MaxPuntos)
        {
         x1=x0+h*(-y0-z0);
         y1=y0+h*(x0 + 0.2*y0);
         z1=z0+h*(0.2 + z0*(x0-5.7));
         x0=x1;
         y0=y1;
         z0=z1;

         glPushMatrix();
         glTranslatef(x0, y0, z0);
         glutSolidSphere(0.20, 12.0, 12.0);
         glPopMatrix();

         Puntos=Puntos+1;
        }
    glFlush ();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
```

```
   glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 200.0);
   glMatrixMode (GL_MODELVIEW);
}

/* ARGSUSED1 */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (600, 600);
   glutInitWindowPosition (0, 0);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

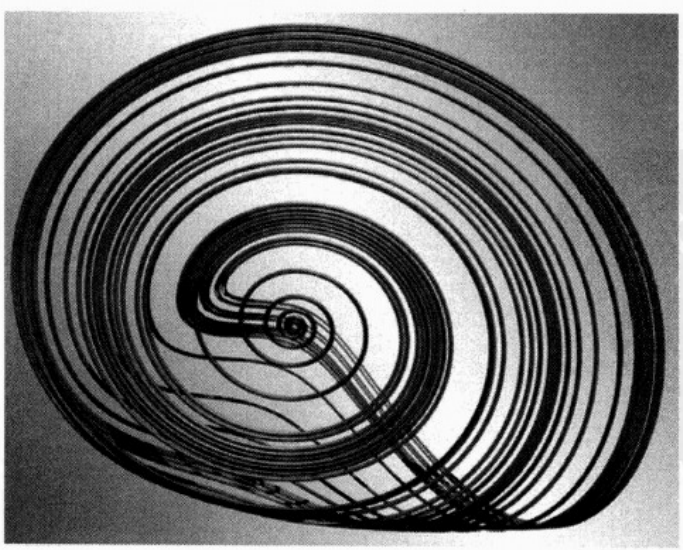


图 8.9　罗斯洛吸引子之二

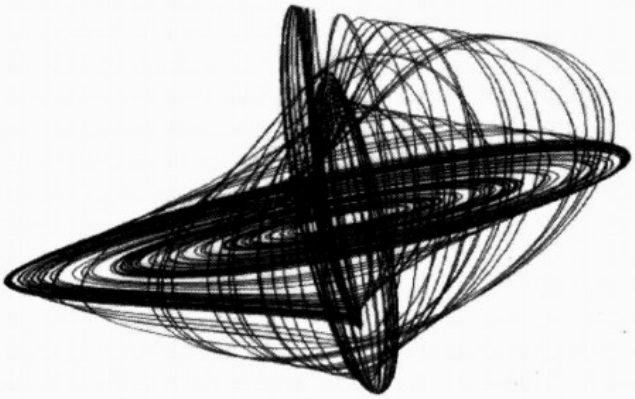


图 8.10　罗斯洛吸引子之三

8.3　杜玢吸引子

杜玢（Duffing）型奇异吸引子来源于杜玢非线性微分方程组

$$\left.\begin{aligned}\dot{x} &= y\\ \dot{y} &= x - x^5 - ay + b\cos(wt)\end{aligned}\right\} \tag{8.5}$$

它常常用来作为经典混沌系统的一个例子，其中典型的常数是 $a=0.25$，$b=0.3$，$w=1$。下面四张图（图 8.11～图 8.14）是 a,b,w 三个常数不同组合得到的。

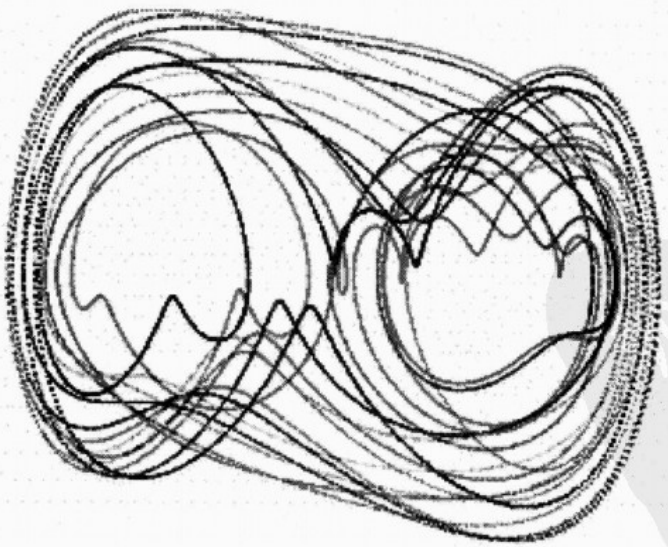


图 8.11　杜玢型奇异吸引子之一

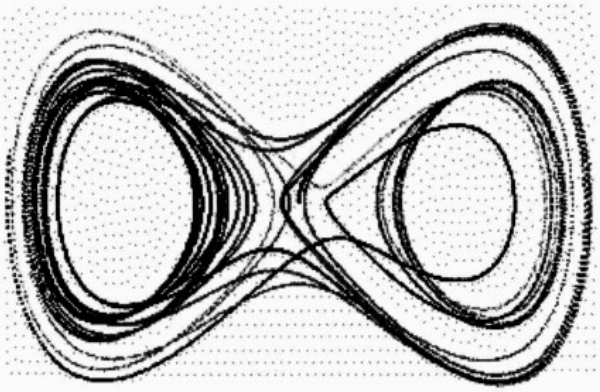


图 8.12　杜玢型奇异吸引子之二

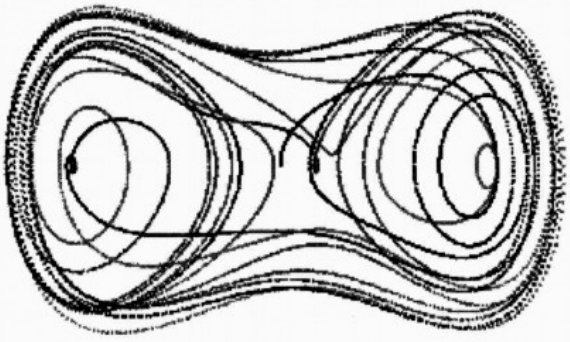


图 8.13　杜玢型奇异吸引子之三

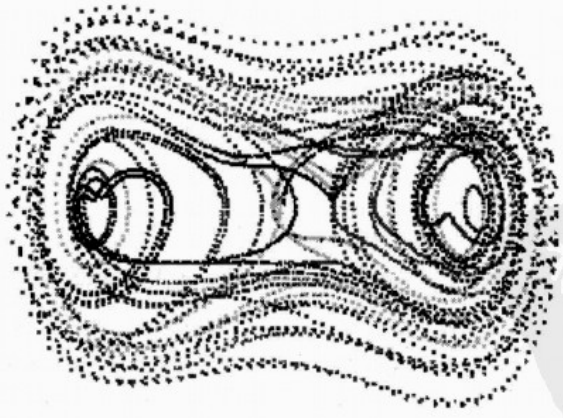


图 8.14　杜玢型奇异吸引子之四

8.4　二元多项式吸引子

以上介绍的各种吸引子都是基于非线性微分方程组的。下面引入另一类吸引子，它

是基于二元多项式方程组而产生的。

8.4.1　亥农吸引子

法国天文学家米歇尔。亥农（Michel Hénon）于 1976 年提出他的二元二次型数学模型为

$$\left.\begin{aligned} X_{n+1} &= 1 + aX_n^2 + bY_n \\ Y_{n+1} &= X_n \end{aligned}\right\} \tag{8.6}$$

这是二维空间的二次映射，式中常数 a 和 b 是控制参数。亥农用的参数值是 $a=-1.4$, $b=0.3$。亥农映射很特别，在每次的迭代后，初始点集合最终压缩成覆盖 XY-平面区域的面积是定常的：通过 bY_n 项，迭代后出现的面积是前一次迭代面积的 30%。其他的 b 值也能应用，但不是所有的值都能产生混沌效果。亥农吸引子的图形见图 8.15，这里我们做了浮雕处理。

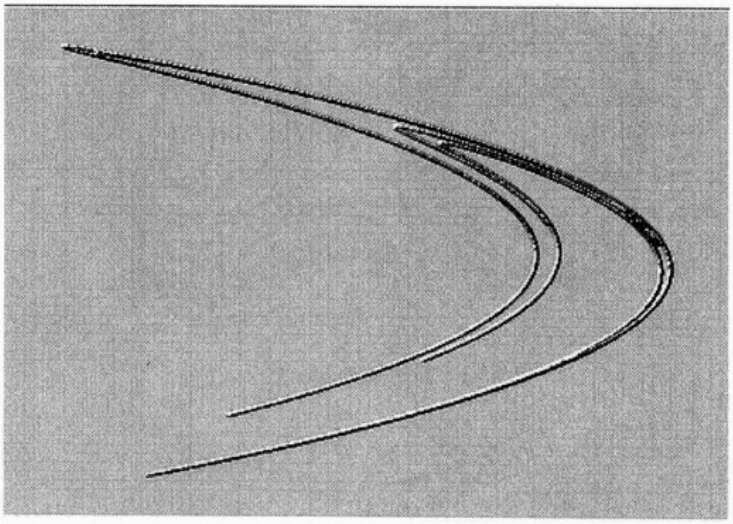


图 8.15　亥农吸引子

模型（8.6）的一般形式是

$$\left.\begin{aligned} X_{n+1} &= a_1 + a_2X_n + a_3X_n^2 + a_4X_nY_n + a_5Y_n + a_6Y_n^2 \\ Y_{n+1} &= a_7 + a_8X_n + a_9X_n^2 + a_{10}X_nY_n + a_{11}Y_n + a_{12}Y_n^2 \end{aligned}\right\} \tag{8.7}$$

当 $a_1=1$, $a_3=-1.4$, $a_5=0.3$, $a_8=1$，且其他系数均为零时便是公式（8.6）。程序 8.4 是上述模型的 C 语言程序。对更一般的多项式方程组（如方程组 8.8 等）的 C 语言程序，由于篇幅太大，读者可以在附带的光盘中找到（PROG28TC.CPP）。

程序 8.4　二次多项式吸引子。

```
/* STRANGE ATTRACTOR PROGRAM */

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
```

```
#include <conio.h>
#include <math.h>
#include <io.h>
#include <string.h>
#include <ctype.h>

int prev, omax, d, ode,  trd, fth, t, wid, p, twod;
int m, i, i1, i2, o, i3, i4, i5, j, wh, nc, c, rd, cy, bk;

char code[515], q;

double nmax, eps, twopi, seg, ne, x, y, z, w, xe, ye, ze, we, lsum, n, nl;
double n1, n2, xmin, xmax, ymin, ymax, zmin, zmax, wmin, wmax, xnew, ynew;
double znew, wnew, xp, yp, ran, xsave, ysave, zsave, wsave, dlx, dly, dlz;
double dlw, dl2, df, rs, l, mx, my, xl, xh, yl, yh, xa, ya, za, tt, pt, tia;
double xz, yz, dur, d2max, dx, dy, dz, dw, d2, f, th, ph, xrt, xlt, hsf, al;
double sinal, cosal, dum, sw, sh, x0, y0;
double xs[500], ys[500], zs[500], ws[500], a[505], v[100], xy[5], xn[5];
union REGS regs;

void initialize(void); /* initialize */
void set_parameters(void); /* set parameters */
void iterate_equations(void); /* iterate equations */
void display_results(void); /* display_results */
void test_results(void); /* test_results */
void get_coefficients(void); /* get_coefficients */
void shuffle_random_numbers(void); /* shuffle_random_numbers */
void calculate_lyapunov_exponent(void); /* calculate_lyapunov_exponent */
void resize_screen(void); /* resize_screen */
void calculate_fractal_dimension(void); /* calculate_fractal_dimension */
void get_dimension_and_order(void); /* get_dimension_and_order */

main()
{
    prev = 5;                  /* plot versus fifth previous iterate */
    nmax = 11000;               /* maximum number of iterations */
    omax = 5;                  /* maximum order of polynomial */
    d = 2;                     /* dimension of system */
    eps = .1;                  /* step size for ode */
    ode = 0;                   /* system is map */
    trd = 1;                   /* display third dimension as shadow */
    fth = 2;                   /* display fourth dimension as colors */
    twopi = 6.28318530717959;   /* a useful constant (2 pi) */
    srand(time(0));             /* reseed random number generator */
    t = 1;
    while (t) {
        switch (t) {
        case 1: initialize();  /* initialize */
        case 2: set_parameters();  /* set parameters */
        case 3: iterate_equations();  /* iterate equations */
```

```
        case 4: display_results();  /* display results */
        case 5: test_results();  /* test results */
        }
    }
    closegraph();
    return(0);
}

void initialize() /* initialize */
{
    int graphdriver = DETECT, graphmode, error_code;
    initgraph(&graphdriver, &graphmode, "");     /* set graphics mode */
    wid = 80;                  /* number of text columns */
    cleardevice();
    gotoxy(wid / 2 - 6, 13);
    printf("searching...");
}

void set_parameters() /* set parameters */
{
    x = .05;  y = .05;  z = .05;    w = .05;
    xe = x + .000001;   ye = y; ze = z; we = w;
    get_coefficients();                /* get coefficients */
    t = 3; p = 0; lsum = 0; n = 0; nl = 0; n1 = 0; n2 = 0; xmin = 1000000;
    xmax = -xmin;   ymin = xmin; ymax = xmax; zmin = xmin; zmax = xmax;
wmin = xmin; wmax = xmax;
    twod = _rotl(1, d);
}

void iterate_equations() /* iterate equations */
{
    m = 1;
    xy[1] = x;  xy[2] = y;  xy[3] = z;  xy[4] = w;
    for (i = 1; i <= d; i++) {
      xn[i] = a[m];
      m = m + 1;
        for (i1 = 1; i1 <= d; i1++) {
          xn[i] = xn[i] + a[m] * xy[i1];
      m = m + 1;
         for (i2 = i1; i2 <= d; i2++) {
          xn[i] = xn[i] + a[m] * xy[i1] * xy[i2];
          m = m + 1;
         for (i3 = i2; o > 2 && i3 <= d; i3++) {
           xn[i] = xn[i] + a[m] * xy[i1] * xy[i2] * xy[i3];
           m = m + 1;
              for (i4 = i3; o > 3 && i4 <= d; i4++) {
            xn[i] = xn[i] + a[m] * xy[i1] * xy[i2] * xy[i3] * xy[i4];
            m = m + 1;

                for (i5 = i4; o > 4 && i5 <= d; i5++) {
```

```
                    xn[i] = xn[i] + a[m] * xy[i1] * xy[i2] * xy[i3] * xy[i4]
                    * xy[i5];
                     m = m + 1;
                    }}}}}
    }
    xnew = xn[1];
    ynew = xn[2];
    znew = xn[3];
    wnew = xn[4];
    n = n + 1;
    m = m - 1;
}

void display_results() /* display results */
{
    if (n >= 100 && n <= 1000) {
        if (x < xmin) xmin = x;       if (x > xmax) xmax = x;
        if (y < ymin) ymin = y;       if (y > ymax) ymax = y;
        if (z < zmin) zmin = z;       if (z > zmax) zmax = z;
        if (w < wmin) wmin = w;       if (w > wmax) wmax = w;
    }
    if ((int)n == 1000) resize_screen();  /* resize the screen */
    xs[p] = x;   ys[p] = y;  zs[p] = z;  ws[p] = w;
    p = (p + 1) % 500;
    i = (p + 500 - prev) % 500;
    xp = x;  yp = y;

    if (n >= 1000 && xp > xl && xp < xh && yp > yl && yp < yh) {
        putpixel(sw * xp - x0, sh * yp - y0, 15); /* plot point on screen */
    }
}

void test_results() /* test results */
{
    if (fabs(xnew) + fabs(ynew) + fabs(znew) + fabs(wnew) > 1000000) t = 2;
    calculate_lyapunov_exponent();     /* calculate lyapunov exponent */
    calculate_fractal_dimension();     /* calculate fractal dimension */
    if (n >= nmax) {    /* strange attractor found */
        t = 2;
    }
    if (fabs(xnew - x) + fabs(ynew - y) + fabs(znew - z) + fabs(wnew
- w) < .000001) t = 2;
    if (n > 100 && l < .005) t = 2; /* limit cycle */

    if (kbhit()) q = getch(); else q = 0;
    if (q) t=0;  /* exit program */
    x = xnew;    y = ynew;    z = znew;    w = wnew;
}

void get_coefficients() /* get coefficients */
```

```
{
    o = 2 + (int)floor((omax - 1) * (float)rand() / 32768.0);
    code[0] = 59 + 4 * d + o + 8 * ode;
    if (ode > 1) code[0] = 87 + ode;
    get_dimension_and_order();              /* get value of m */
    for (i = 1; i <= m; i++) {  /* construct code */
        shuffle_random_numbers();       /* shuffle random numbers */
        code[i] = 65 + (int)floor(25 * ran);
    }
    code[m + 1] = 0;
    for (i = 1; i <= m; i++) {  /* convert code to coefficient values */
        a[i] = (code[i] - 77) / 10.0;
    }
}

void shuffle_random_numbers() /* shuffle random numbers */
{
    if (v[0] == 0) for (j = 0; j <= 99; j++) {v[j] = (float)rand() / 32768.0;}
    j = (int)floor(100 * ran);
    ran = v[j];
    v[j] = (float)rand() / 32768.0;
}

void calculate_lyapunov_exponent() /* calculate lyapunov exponent */
{
    xsave = xnew;    ysave = ynew;   zsave = znew;   wsave = wnew;
    x = xe;  y = ye; z = ze; w = we;
     n = n - 1;
    iterate_equations();                /* reiterate equations */
    dlx = xnew - xsave; dly = ynew - ysave;
    dlz = znew - zsave; dlw = wnew - wsave;
    dl2 = dlx * dlx + dly * dly + dlz * dlz + dlw * dlw;
    if (dl2 > 0) {            /* check for division by zero */
        df = 1e12 * dl2;
        rs = 1 / sqrt(df);
        xe = xsave + rs * (xnew - xsave);
        ye = ysave + rs * (ynew - ysave);
        ze = zsave + rs * (znew - zsave);
        we = wsave + rs * (wnew - wsave);
        xnew = xsave; ynew = ysave;    znew = zsave;    wnew = wsave;
        lsum = lsum + log(df);
        nl = nl + 1;
        l = .721347 * lsum / nl;
        if (n > 1000 && (int)n % 10 == 0) {
            gotoxy(wid - 4, 1);
            printf("%5.2f", l);
        }
    }
}

void resize_screen() /* resize the screen */
```

```
{
    int poly[8];
    setcolor(wh);
    if (xmax - xmin < .000001) {
        xmin = xmin - .0000005;
        xmax = xmax + .0000005;
    }
    if (ymax - ymin < .000001) {
        ymin = ymin - .0000005;
        ymax = ymax + .0000005;
    }
    if (zmax - zmin < .000001) {
        zmin = zmin - .0000005;
        zmax = zmax + .0000005;
    }
    if (wmax - wmin < .000001) {
        wmin = wmin - .0000005;
        wmax = wmax + .0000005;
    }
    mx = .1 * (xmax - xmin);my = .1 * (ymax - ymin);
    xl = xmin - mx;  xh = xmax + mx;  yl = ymin - my;  yh = ymax + 1.5 * my;
    sw = (getmaxx() + 1) / (xh - xl);
    sh = (getmaxy() + 1) / (yl - yh);
    x0 = sw * xl;    y0 = sh * yh;
    cleardevice();
    yh = yh - .5 * my;
    xa = (xl + xh) / 2;  ya = (yl + yh) / 2;
    rectangle(0, getmaxy() / 24, getmaxx(), getmaxy());
    tt = 3.1416 / (xmax - xmin);pt = 3.1416 / (ymax - ymin);
    gotoxy(1, 1);
    if (strlen(code) < wid - 18)
        printf(code);
    else {
        printf("%*.*s...", wid - 23, wid - 23, code);
    }
    gotoxy(wid - 17, 1);
    printf("f =");
    gotoxy(wid - 7, 1);
    printf("l = ");

    tia = .05;                  /* tangent of illumination angle */
    xz = -tia * (xmax - xmin) / (zmax - zmin);
    yz = tia * (ymax - ymin) / (zmax - zmin);
}

void calculate_fractal_dimension() /* calculate fractal dimension */
{
    if (n >= 1000) {            /* wait for transient to settle */
```

```
            if ((int)n == 1000) {
                d2max = pow(xmax - xmin, 2);
                d2max = d2max + pow(ymax - ymin, 2);
                d2max = d2max + pow(zmax - zmin, 2);
                d2max = d2max + pow(wmax - wmin, 2);
            }
            j = (p + 1 + (int)floor(480 * (float)rand() / 32768.0)) % 500;
            dx = xnew - xs[j];        dy = ynew - ys[j];
            dz = znew - zs[j];        dw = wnew - ws[j];
            d2 = dx * dx + dy * dy + dz * dz + dw * dw;
            if (d2 < .001 * twod * d2max) n2 = n2 + 1;
            if (d2 <= .00001 * twod * d2max) {
                n1 = n1 + 1;
                f = .434294 * log(n2 / (n1 - .5));
                gotoxy(wid - 14, 1);
                printf("%5.2f", f);
            }
        }
    }

    void get_dimension_and_order() /* get dimension and order */
    {
        o = 2 + (code[0] - 65) % 4;
        m = 1;
        for (i = 1; i <= d; i++) {m = m * (o + i);}
    }
```

当 $a_i(i=1,2,\cdots,12)$ 分别取 0.323，−0.041，0.472，−1.051，0.089000000001，−0.821，−0.747，0.587，0.788，−0.329，1.101，1.002 时，可得到图 8.16。

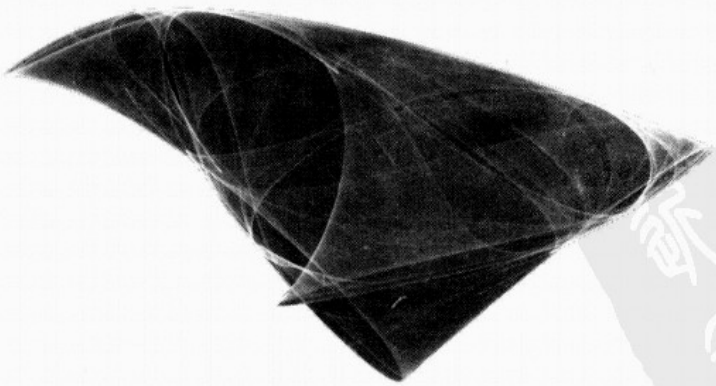


图 8.16　二元二次多项式吸引子之一

当 $a_i(i=1,2,\cdots,12)$ 分别取 0.085，−0.665，0.342，0.817，0.851，−0.029，0.722，−0.143，−0.836，0.741，−0.527，0.933 时，可得到图 8.17。

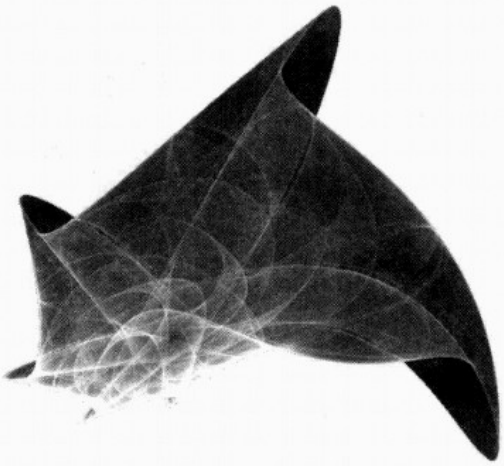


图 8.17　二元二次多项式吸引子之二

当 $a_i(i=1,2,\cdots,12)$ 分别取 −0.28, 0.21, 1.051, −0.66, 0.051, −0.925, 0.926, −0.456, −0.308, 1.184, −0.706, −0.259 时，可得到图 8.18。

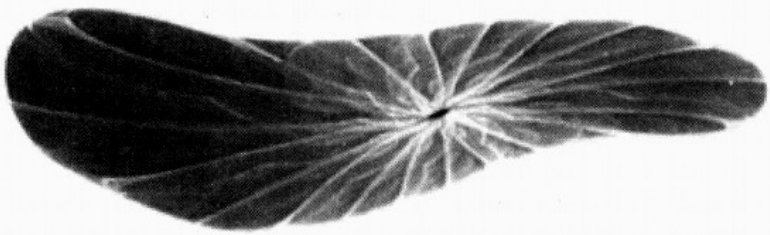


图 8.18　二元二次多项式吸引子之三

8.4.2　二元高次多项式型

可以把二次型方程组推广到三次无序映射：

$$\left.\begin{aligned} X_{n+1} &= a_1 + a_2X_n + a_3X_n^2 + a_4X_n^3 + a_5X_n^2Y_n + a_6X_nY_n + a_7X_nY_n^2 \\ &\quad + a_8Y_n + a_9Y_n^2 + a_{10}Y_n^3 \\ Y_{n+1} &= a_{11} + a_{12}X_n + a_{13}X_n^2 + a_{14}X_n^3 + a_{15}X_n^2Y_n + a_{16}X_nY_n + a_{17}X_nY_n^2 \\ &\quad + a_{18}Y_n + a_{19}Y_n^2 + a_{20}Y_n^3 \end{aligned}\right\} \tag{8.8}$$

方程（8.8）含有 20 个系数，这将极大地增加状态变化的数量。如果是四次映射，将有 30 个系数，五次映射有 42 个系数，n 次映射将多达 $(n+1)(n+2)$ 个系数。

三次映射的程序设计可以从二次的程序演变而来，请参看光盘中给出的程序 PROG28TC.CPP。

当 $a_i(i=1,2,\cdots,20)$ 分别取 0.27, 1.079, –0.721, –1.17, 1.044, –0.887, –0.446, –0.462, –0.806, 1.014, 0.113, –1.101, –0.75, –1.168, –0.948, 0.057, 0.077, –0.817, –0.995, 0.911 时，可得到图 8.19。

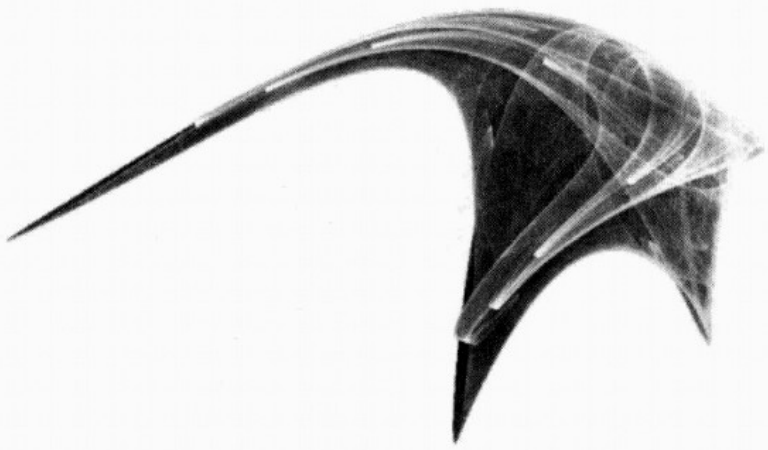


图 8.19　二元三次多项式吸引子之一

当 $a_i(i=1,2,\cdots,20)$ 分别取 0.838, 0.58, –1.072, –1.199, 0.296, –0.566, –1.041, 0.611, 0.251, 0.77, –0.347, –0.821, 0.782, 0.561, 0.995, 0.535, 0.716, 1.032, 0.488, –0.278 时，可得到图 8.20。

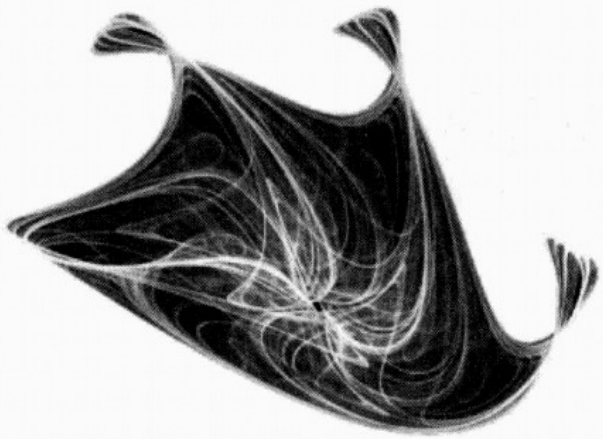


图 8.20　二元三次多项式吸引子之二

也可以把二次型方程组推广到四次无序映射，二元四次多项式的一般形式为

$$\left.\begin{aligned} X_{n+1} &= a_1 + a_2X_n + a_3{X_n}^2 + a_4{X_n}^3 + a_5{X_n}^4 + a_6{X_n}^3Y_n + a_7{X_n}^2{Y_n}^2 \\ &\quad + a_8{X_n}^2Y_n + a_9X_nY_n + a_{10}X_n{Y_n}^2 + a_{11}X_n{Y_n}^3 + a_{12}Y_n + a_{13}{Y_n}^2 + a_{14}{Y_n}^3 + a_{15}{Y_n}^4 \\ Y_{n+1} &= a_{16} + a_{17}X_n + a_{18}{X_n}^2 + a_{19}{X_n}^3 + a_{20}{X_n}^4 + a_{21}{X_n}^3Y_n + a_{22}{X_n}^2{Y_n}^2 \\ &\quad + a_{23}{X_n}^2Y_n + a_{24}X_nY_n + a_{25}X_n{Y_n}^2 + a_{26}X_n{Y_n}^3 + a_{27}Y_n + a_{28}{Y_n}^2 + a_{29}{Y_n}^3 + a_{30}{Y_n}^4 \end{aligned}\right\} \quad (8.9)$$

当 $a_i(i=1,2,\cdots,30)$ 分别取 0.916, –0.89, –1.046, 0.958, –0.398, 0.415, –0.754, 0.358, 0.766, 0.435, 0.932, 1.051, –0.789, –0.37, –0.839, 0.264, –0.913, –1.191, 1.045, 0.625, 0.577, –0.613, 0.477, 000000001, 0.307, 0.69, –0.193, –0.265, –0.434, 0.86 时，可以得到图 8.21。

当 $a_i(i=1,2,\cdots,30)$ 分别取 0.435, –0.613, 0.337, –0.87, –0.666, –0.24, 0.981, –0.049,

0.425, 0.43,0.218, –0.579, –0.1, –0.525, 1.119, –0.351, –0.027, –0.865, 0.947, 1.024, –1.072, –0.405, 0.825, –0.445, –1.086, –0.507, –0.343, 1.114, –0.423, –0.324 时，可以得到图 8.22。

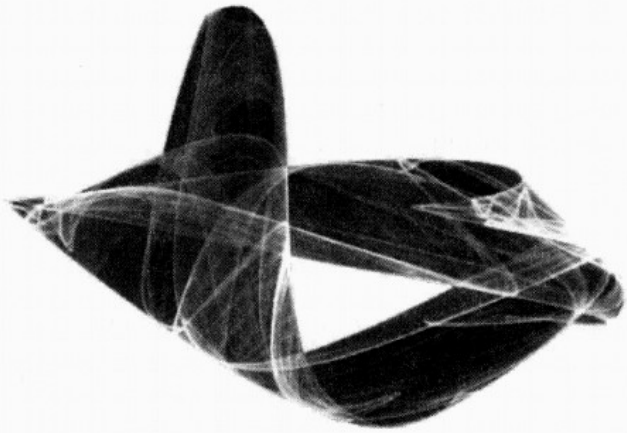


图 8.21　二元四次多项式吸引子之一

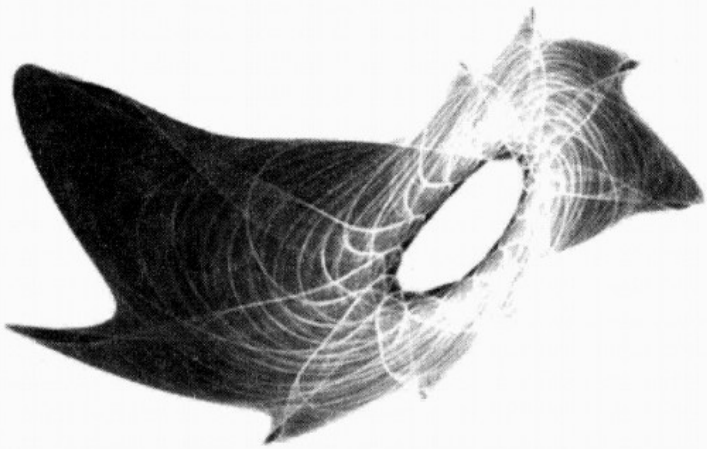


图 8.22　二元四次多项式吸引子之二

二元五次多项式的一般形式为

$$\left.\begin{aligned}
X_{n+1} &= a_1 + a_2X_n + a_3{X_n}^2 + a_4{X_n}^3 + a_5{X_n}^4 + a_6{X_n}^5 + a_7{X_n}^4Y_n + a_8{X_n}^3Y_n \\
&\quad + a_9{X_n}^3{Y_n}^2 + a_{10}{X_n}^2Y_n + a_{11}{X_n}^2{Y_n}^2 + a_{12}{X_n}^2{Y_n}^3 + a_{13}X_nY_n + a_{14}X_n{Y_n}^2 \\
&\quad + a_{15}X_n{Y_n}^3 + a_{16}X_n{Y_n}^4 + a_{17}Y_n + a_{18}{Y_n}^2 + a_{19}{Y_n}^3 + a_{20}{Y_n}^4 + a_{21}{Y_n}^4 \\
Y_{n+1} &= a_{22} + a_{23}X_n + a_{24}{X_n}^2 + a_{25}{X_n}^3 + a_{26}{X_n}^4 + a_{27}{X_n}^5 + a_{28}{X_n}^4Y_n \\
&\quad + a_{29}{X_n}^3Y_n + a_{30}{X_n}^3{Y_n}^2 + a_{31}{X_n}^2Y_n + a_{32}{X_n}^2{Y_n}^2 + a_{33}{X_n}^2{Y_n}^3 + a_{34}X_nY_n \\
&\quad + a_{35}X_n{Y_n}^2 + a_{36}X_n{Y_n}^3 + a_{37}X_n{Y_n}^4 + a_{38}Y_n + a_{39}{Y_n}^2 + a_{40}{Y_n}^3 + a_{41}{Y_n}^4 + a_{42}{Y_n}^4
\end{aligned}\right\} \quad (8.10)$$

当 $a_i(i=1,2,\cdots,42)$ 分别取 0.293, –1.138, –0.203, 0.685, 0.133, 1.1, 1.022, 1.187, 0.77, 1.147, 1.039, –0.307, 0.155, 0.696, 0.889, 0.598, –0.0989999999999999, –0.468, 1.011, –0.793,

0.177, 0.198, –0.797, 0.847, –0.475, 0.263, 0.927, 0.641, –1.05, 0.198, –0.286, –0.038, 0.808, 0.244, –0.17, 0.013, 1.042, –0.914, –0.677, –0.531, –0.684, –0.888 时，可以得到图 8.23。

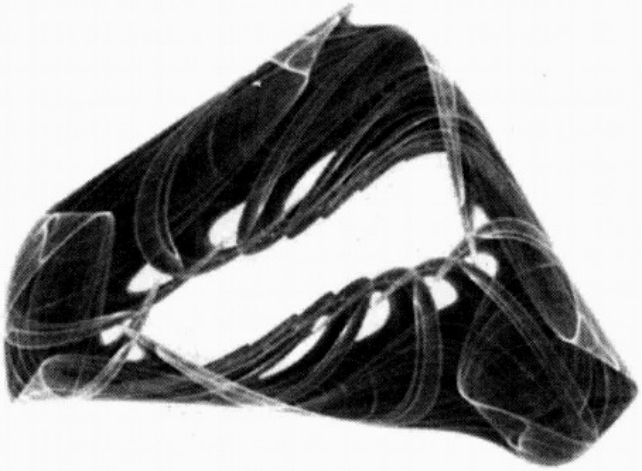


图 8.23　二元五次多项式吸引子之一

当 $a_i(i=1,2,\cdots,42)$ 分别取 0.607, 0.853, –0.363, –0.377, –0.447, 0.259, –0.712, 0.494, 0.075, –0.891, –0.283, –0.481, 1.139, –0.93, 0.886, –0.889, 0.147, 0.217, 0.042, 0.208, 0.944, –0.778, –0.4, 0.981, –0.21, –0.485, –1.163, –1.09, –0.98, 0.765, –0.795, –0.02, 0.356, 0.109, 0.803, 0.722, –0.955, –0.256, 1.088, 1.177, 0.244, 0.783 时，可以得到图 8.24。



图 8.24　二元五次多项式吸引子之二

8.5　三元多项式吸引子

8.5.1　二次多项式映射

我们容易把上述二维情况推广到三维空间，即在 X 和 Y 平面垂直伸出一个 Z 轴来，这就是著名的 Sprott 型多项式。

$$\left.\begin{aligned} x' &= a_0 + a_1x + a_2x^2 + a_3xy + a_4xz + a_5y + a_6y^2 + a_7yz + a_8z + a_9z^2 \\ y' &= b_0 + b_1x + b_2x^2 + b_3xy + b_4xz + b_5y + b_6y^2 + b_7yz + b_8z + b_9z^2 \\ z' &= c_0 + c_1x + c_2x^2 + c_3xy + c_4xz + c_5y + c_6y^2 + c_7yz + c_8z + c_9z^2 \end{aligned}\right\} \quad (8.11)$$

方程组（8.11）共有 30 个参数。当参数依次为

0.862, –1.147, 0.01, 0.241, –0.85, –0.174, 1.193, –0.572, 0.772, 0.147; –0.049, 0.427, –0.103, –0.402, 1.13, 0.859, –0.642, –0.649, –1.074, –0.636 和 –0.706, 0.315, –0.125, 1.193, 0.533, –0.091, 0.778, –1.199, –0.112, 0.025 时，画得图形见彩色图片 10（左）。

当参数依次为

0.021, 1.182, –1.183, 0.128, –1.12, –0.641, –1.152, –0.834, –0.97, 0.722； 0.243038, –0.825, –1.2, –0.835443, –0.835443, –0.364557, 0.458, 0.622785, –0.394937, –1.032911; 和–0.455696, 0.673, 0.915, –0.258228, –0.495, –0.264, –0.432, –0.416, –0.877, –0.3 时，画得图 8.25。图 8.26 是其他参数对应的图像。

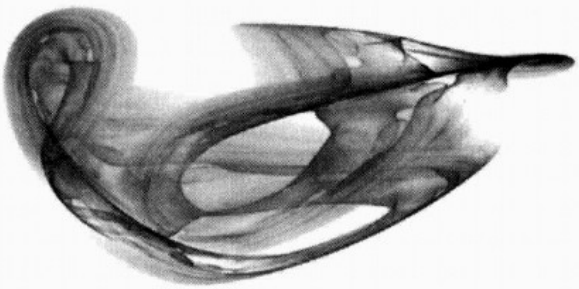


图 8.25　三元二次映射吸引子之一

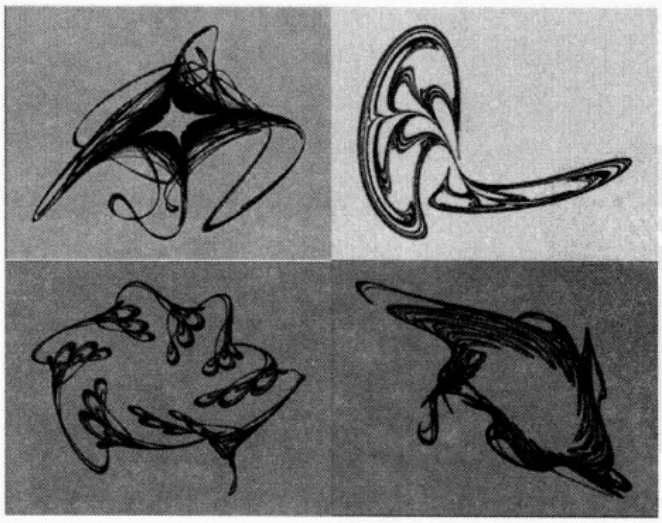


图 8.26　三元二次映射吸引子之二

若对图 8.27 做如下变换：

$$\left.\begin{aligned} X_p &= -X\sin q + Y\cos q \\ Y_p &= -X\sin q\cos f - Y\sin q\cos f + Z\sin f \end{aligned}\right\} \tag{8.12}$$

则可把原图形投影到 XY 平面上，式中 q 和 f 是角度。当然也可投影到 XZ 或 YZ 平面上。

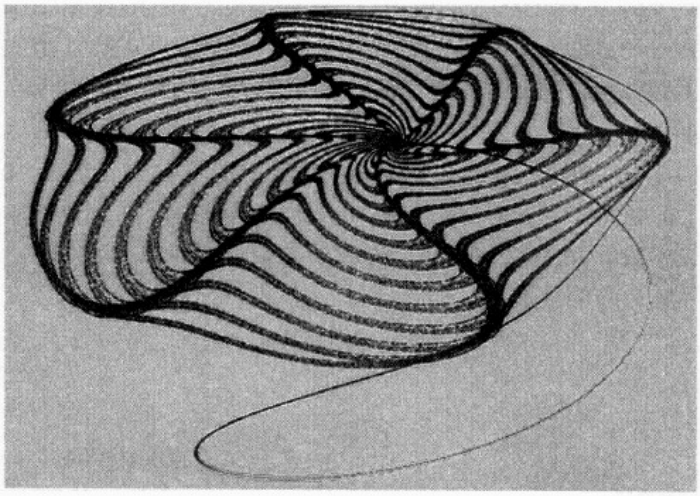


图 8.27　三元二次映射吸引子之三

图 8.28 自左向右分别为投影到 YZ 和 XZ 平面上的二维图形。

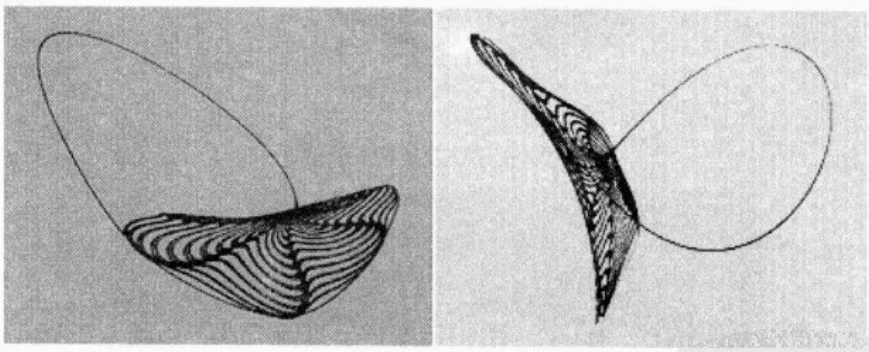


图 8.28　YZ 和 XZ 平面上的投影图

8.5.2　三类参数量相异的二次型

A 型。这是最简单的仅含三个参数的方程组，其形式如下：

$$\left.\begin{aligned} x' &= p_1 + y - zy \\ y' &= p_2 + z - xz \\ z' &= p_3 + x - yx \end{aligned}\right\} \tag{8.13}$$

令三个参数分别为 1.58389, 1.11409, 0.26846, 并得到图形，见图 8.29。

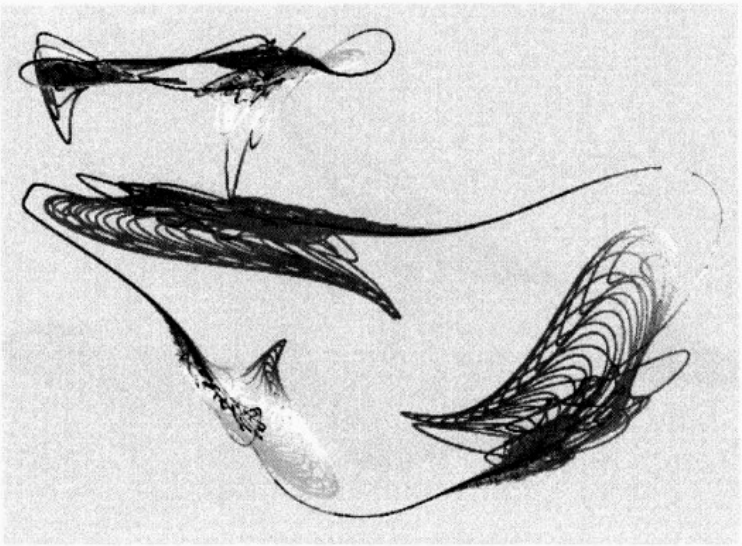


图 8.29　A 型吸引子之一

再令参数为 1.62416, 1.10067, 0.25503, 得到漂亮的彩色图片 10（右）。

B 型。该方程组含六个参数，其形式为

$$\left.\begin{aligned} x' &= p_1 + y - z(p_2 + y) \\ y' &= p_3 + z - x(p_4 + z) \\ z' &= p_5 + x - y(p_6 + x) \end{aligned}\right\} \tag{8.14}$$

令六个参数分别为–0.237, 0.453, 0.486, 0.616, 1.141, 0.483，得到图 8.30。

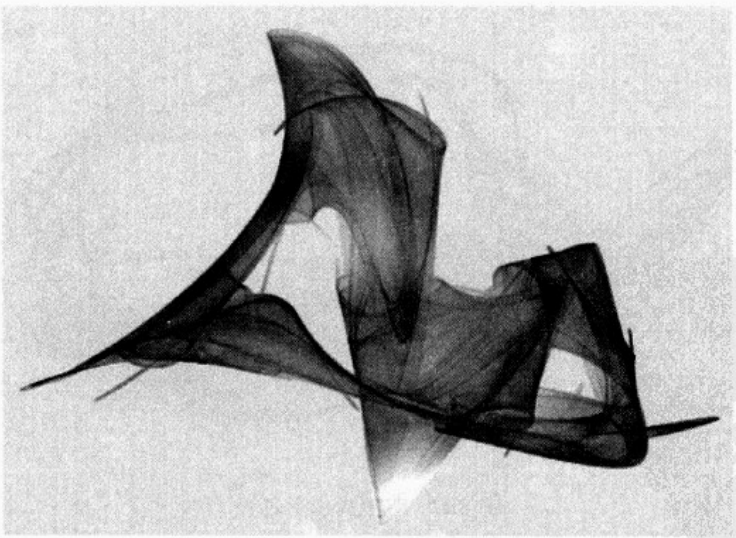


图 8.30　B 型吸引子之一

再令这些参数分别为 0.76421, 0.02091, 0.42685, 0.87959, 1.14846 和 0.68456，则得到图 8.31。

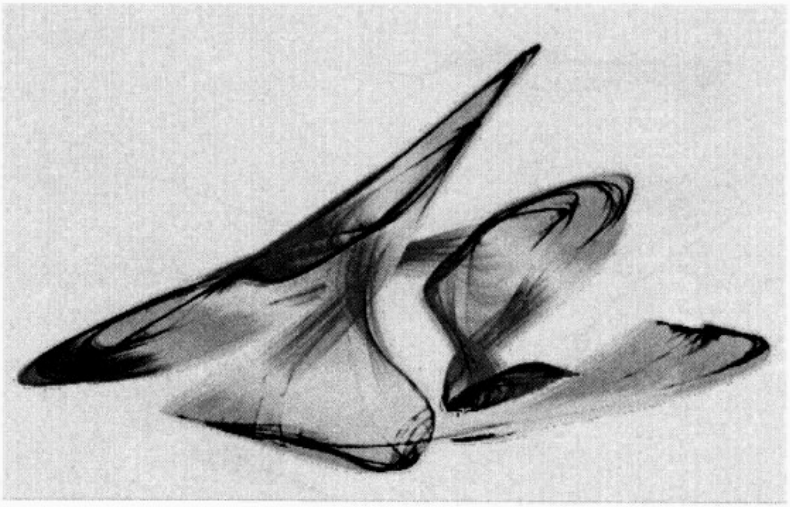


图 8.31　B 型吸引子之二

C 型。这类方程组有十八个参数，形式如下：

$$\left.\begin{aligned} x' &= p_1 + x(p_2 + p_3 x + p_4 y) + y(p_5 + p_6 y) \\ y' &= p_7 + y(p_8 + p_9 y + p_{10} z) + z(p_{11} + p_{12} z) \\ z' &= p_{13} + z(p_{14} + p_{15} z + p_{16} x) + x(p_{17} + p_{18} x) \end{aligned}\right\} \tag{8.15}$$

取参数值为

–0.653165, –0.972152, –0.713924, –0.481, 0.516456, –0.592405, –0.268, 0.827, 0.379747, –0.943, –0.072, 1.2, –0.47, 0.041, 0, 0.914, –0.531646, 0.364557，得到图 8.32。



图 8.32　C 型吸引子之一

取参数值为

0.425316, –0.349367, –0.881013, 0.774684, –0.410127, 0.805063, 0.379747, –0.786, 0.683544, 1.078481, 0.137, 1.16962, 0.410127, –0.318987, –0.531646, –0.394937, 1.15443, –1.001，得到图 8.33。

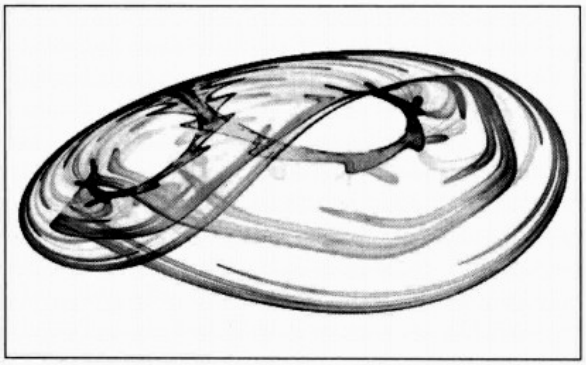


图 8.33　C 型吸引子之二

取参数值为

–0.075949, –0.774684, 0.364557, –0.303797, 0.364557, 1.017722, 0.243038, –0.972152, –0.091139, –0.273418, 0.987342, 0.805063, 0.04557, 0.637975, 0.744304, 0.075949, 0.2127, –1.16962，得到图 8.34。

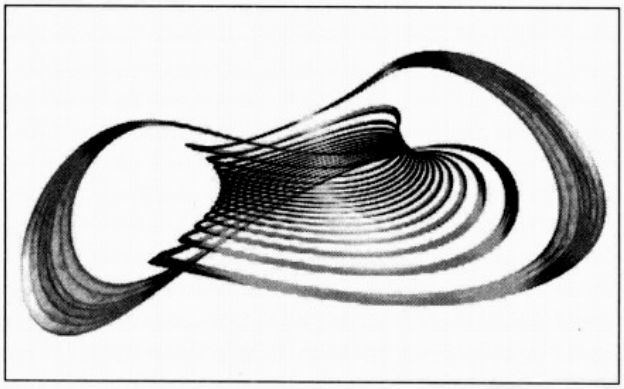


图 8.34　C 型吸引子之三

事实上，多项式方程组的迭代映射是千变万化的，其产生的吸引子更是数不胜数，见图 8.35～图 8.37 和彩色图片 9。在本书附带的光盘中，还有大量的这种图形。

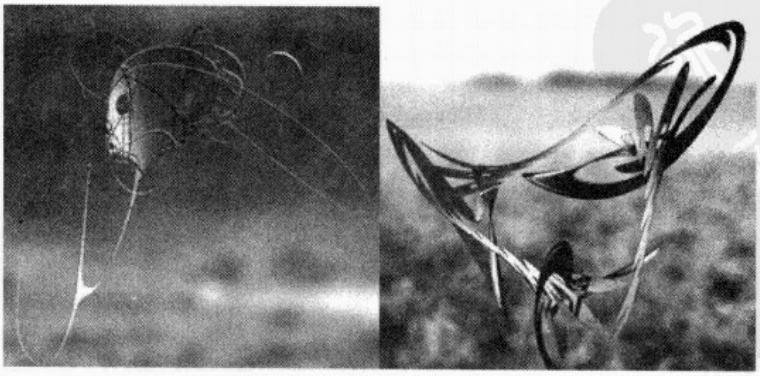


图 8.35　多项式映射吸引子之 A

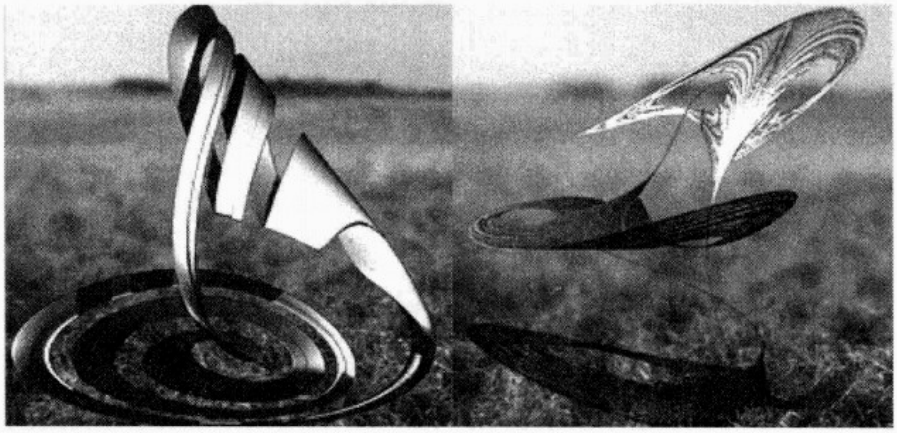


图 8.36　多项式映射吸引子之 B

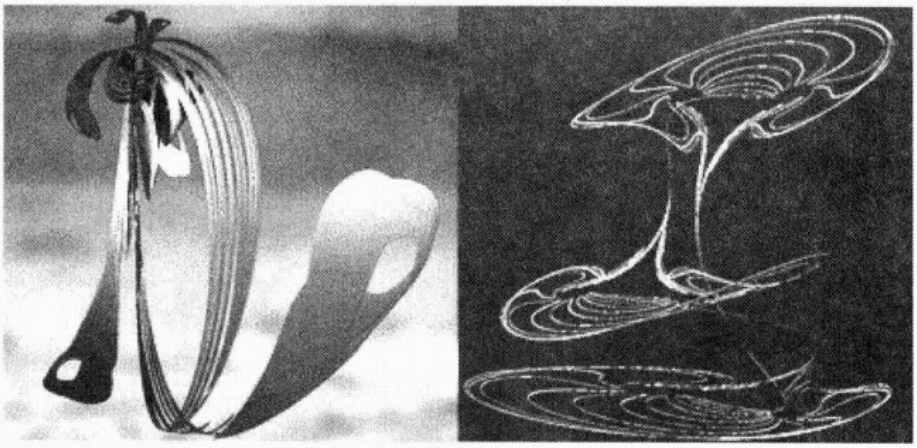


图 8.37　多项式映射吸引子之 C

8.6　皮柯佛奇异吸引子

皮柯佛（Clifford A. Pickover）是美国极具才华的分形学家，著作甚多，影响很大。皮柯佛奇异吸引子分二维和三维两种情况。

二维空间的是三角变换型，方程组如下：

$$\left.\begin{aligned} x' &= \sin ay + c\cos ax \\ y' &= \sin bx + d\cos by \end{aligned}\right\} \tag{8.16}$$

分别令

$$a=1.1,\ b=-1.0,\ c=1.0,\ d=1.5$$

和

$$a=1.5,\ b=-1.8,\ c=1.6,\ d=0.9$$

画得图 8.38。

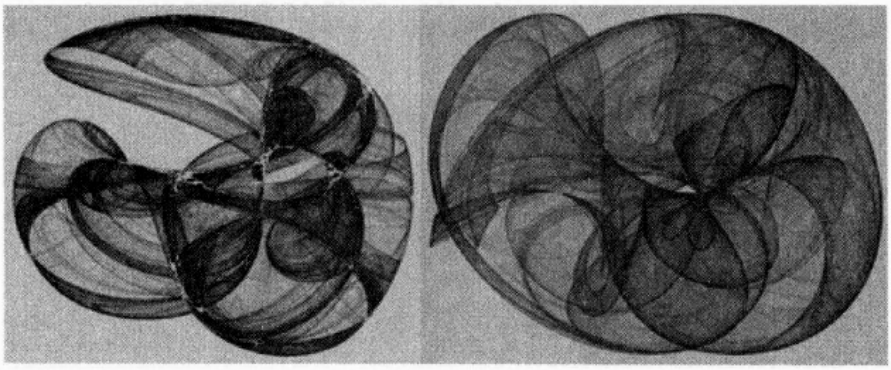


图 8.38　皮克佛吸引子之一

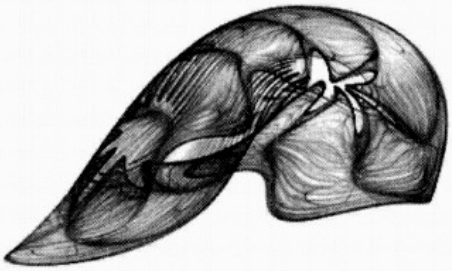
三维空间的也是三角变换型，其方程组如下：

$$\left.\begin{aligned} x' &= \sin ay - z\cos bx \\ y' &= z\sin cx - \cos dy \\ z' &= \sin x \end{aligned}\right\} \tag{8.17}$$

共有四个参数。令 a, b, c, d 分别为

1.405063, 2.012658, 0.56962, 1.537975

得到图 8.39，其图形分上下两部分。



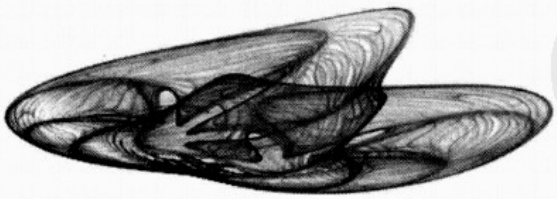


图 8.39　皮克佛吸引子之二

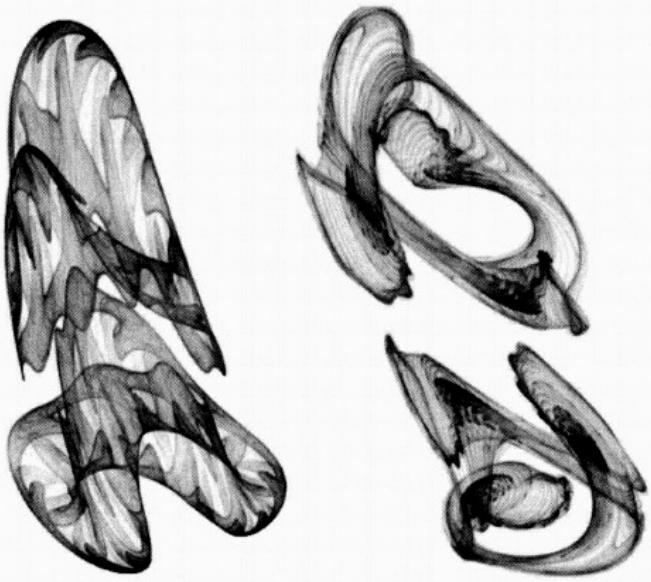
令 a, b, c, d 为

−0.759494, 2.449367, 1.253165, 1.5

和

−1.139241, 3, 1.936709, 1.5

时，分别画得图 8.40 的(a)和(b)两幅图形，其中(a)的图形仿佛是一个带面纱跪在地上的一个人。



(a)　　(b)

图 8.40　皮克佛吸引子之三

图 8.41 是当令 a, b, c, d 为 1.67088607594937, 2.16455696202532, 1.14724050632911, 1.5 时得到。

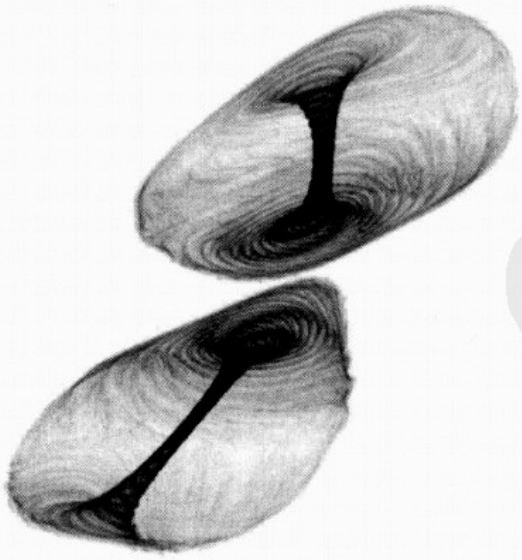


图 8.41　皮克佛吸引子之四

8.7　其他类奇异吸引子

8.7.1　彼得·德·戎(Peter de Jong)型

这也是一种平面上的三角变换型，方程组如下：

$$\left.\begin{aligned} x' &= \sin ay - \cos bx \\ y' &= \sin cx - \cos dy \end{aligned}\right\} \tag{8.18}$$

分别令

$$a = 1.4,\ b = -2.3,\ c = 2.4,\ d = -2.1$$

和

$$a = -2.7,\ b = -0.09,\ c = -0.86,\ d = -2.2$$

得到图 8.42。

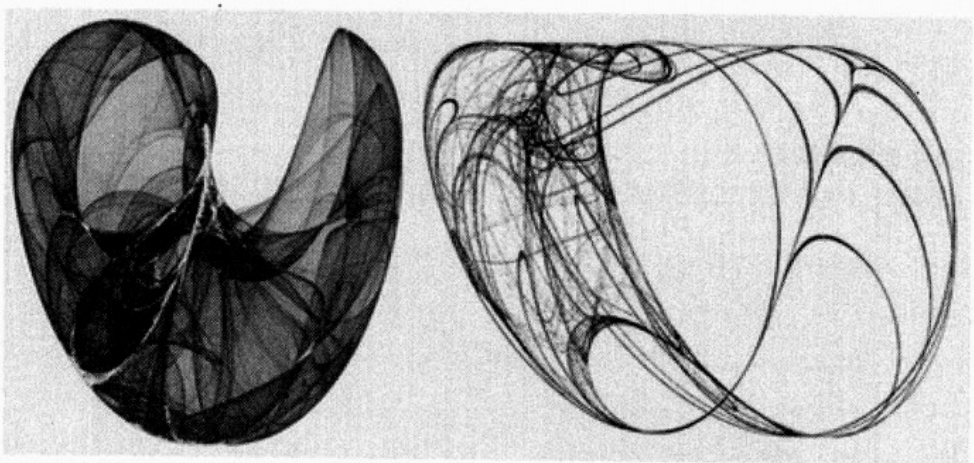


图 8.42　彼得·德·戎吸引子

8.7.2　对称型

这组方程是迈克·菲尔德（Michael Field）和马丁·格路比茨基（Martin Golubitsky）用于演示混沌是如何引出对称来的。这种吸引子的特点是，它由二维转向三维，但第三维的值 Z 却不用于 X 和 Y 的计算。

$$\left.\begin{aligned} & r \in C \\ & r = (x + iy)^d \\ & p = \lambda + \alpha \| r \| + \beta(x \operatorname{Re} r - y \operatorname{Im} r) \\ & x' = px + \gamma \operatorname{Re} r - \omega y \\ & y' = py - \gamma \operatorname{Im} r + \omega x \\ & z' = \| r \| \end{aligned}\right\} \tag{8.19}$$

共有六个参数：$d, \alpha, \beta, \lambda, \gamma$和$\omega$。分别取参数为 19, –2.632, –2.355, 2.373, 0.44, 0.574, 得到图 8.43。

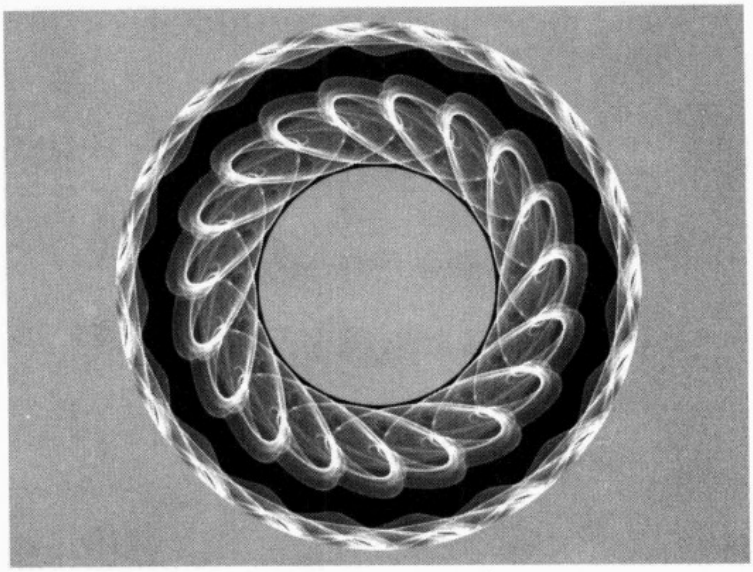


图 8.43　对称型吸引子之一

当分别取参数为

0.64557, 0.417722, 0.025316, 2.35443, 2.316456, 2.43038 时，得到图 8.44。

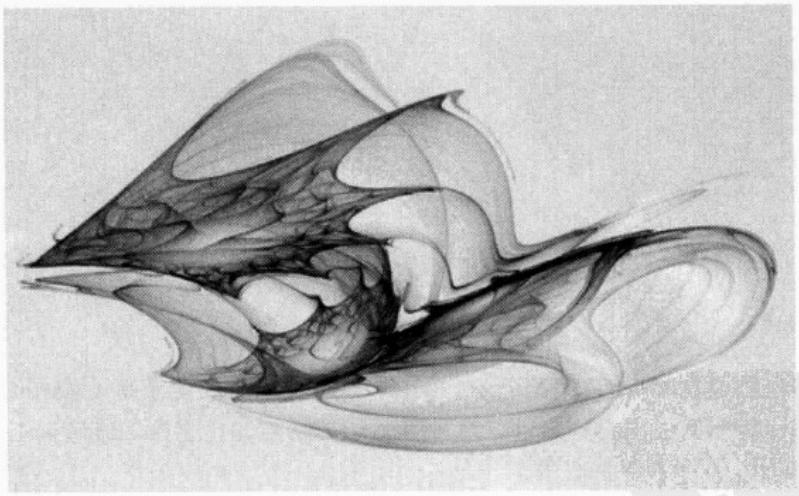


图 8.44　对称型吸引子二

当分别取参数为

0.240506, 0, –0.936709, 1.898734, –2.962025, –2.620253, 1.518987 时，得到图 8.45。

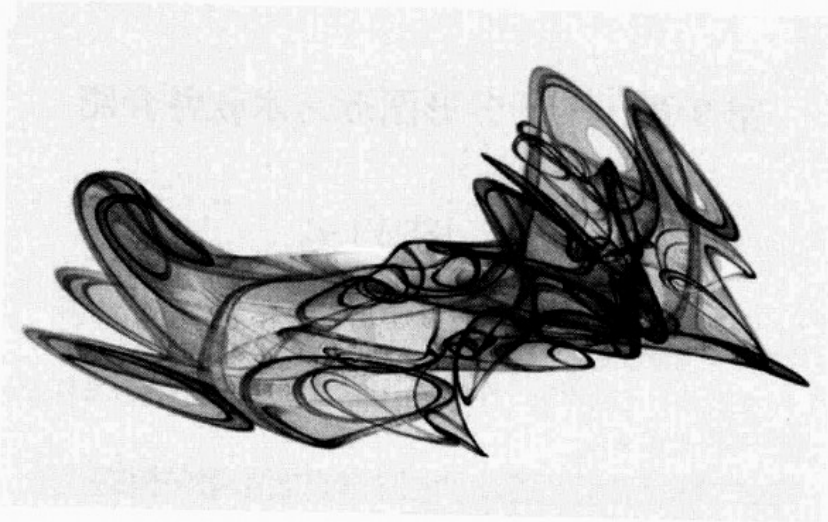


图 8.45　对称型吸引子之三

我们看到，采用不同的计算公式会画出千姿百态的吸引子。问题是，是否随便编一个公式就能产生奇异吸引子呢？可以肯定的是，所编写的方程组必须是与混沌有关。

第 9 章　若干分形图形艺术软件介绍

9.1　IFS V1.41

如前所言，迭代函数系统 IFS 在分形的理论和实践中都起着重要的作用，因此有一个好的 IFS 软件是很有意义的。

IFS V1.41 是我们自己创作的。目的是为使用者提供一个方便且富有艺术性的分形画图工具。软件界面见图 9.1。

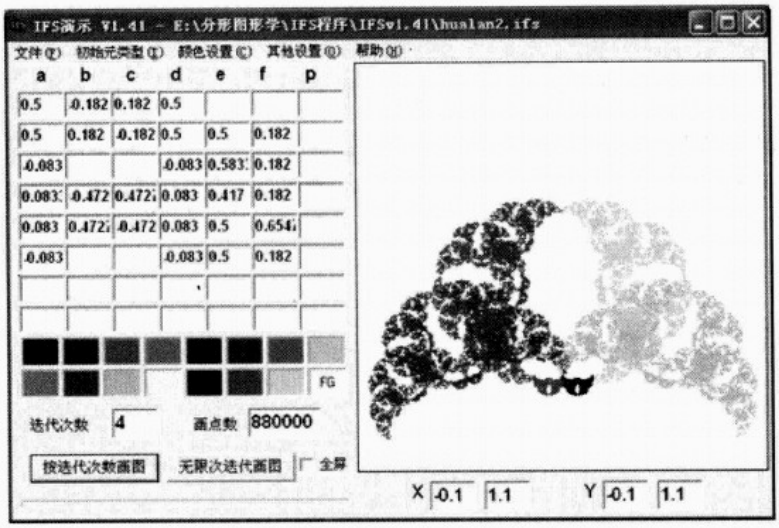


图 9.1　IFS V1.41 页面

页面的左半面上部有画图的内容设置，a，b，c，d，e，f，p 恰是 IFS 码的代码符。下面的网格每一行就是一个仿射变换 W_n，本软件可提供最多八个仿射变换。网格的下面是颜色选择，当选择到某一颜色时，画出来的分形图就是单色的。如果选择白色（默认）时，画面是多色的，且 N 个变换就出现 N 种颜色。如果该 IFS 的吸引子是 S，则每种颜色的图案就表示某个 $W_n(S)$（$n=1, \cdots, N$）。画图时，既可按迭代次数画也可以“无限次迭代”画。

页面的右半面是画图所在，其下方是变量 X 和 Y 的定义域。调整定义域可使画面处于适当位置。如果嫌画面不够大，可点击左面下方的“全屏”处方框，这时，画出的图画会充斥整个屏幕。

使用时，只要在画面的左边于网格中填入相应的 IFS 码，然后按前面所言就可以画出图画了。如某代码是零，则不必填入。如等概率时，也不必填进数字。图 9.1 的分形图就是图 5.18。

本软件画图时，有三种初始设置：正方形、三角形和线段，见图 9.2。

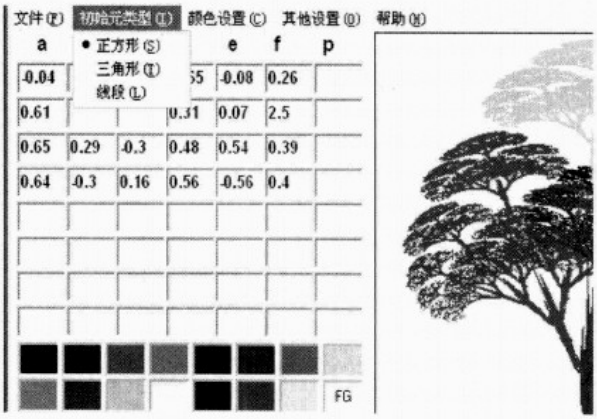


图 9.2　初始元类型设置选择

图 9.3 显示了画蕨叶时，初始类型为三角形且迭代第六次时的图形。

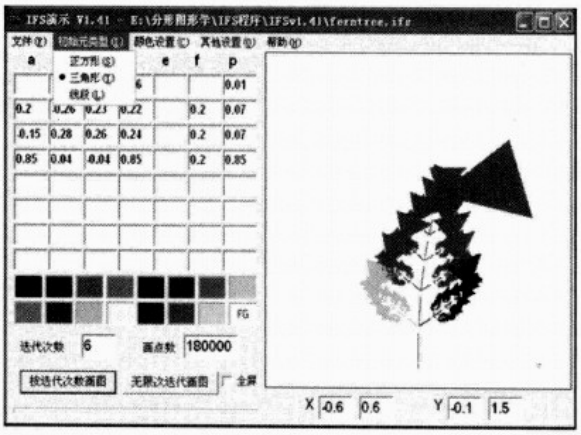


图 9.3　三角形为初值且迭代至六次时的蕨叶图

图案的颜色有单色和彩色两种设置，见图 9.4。

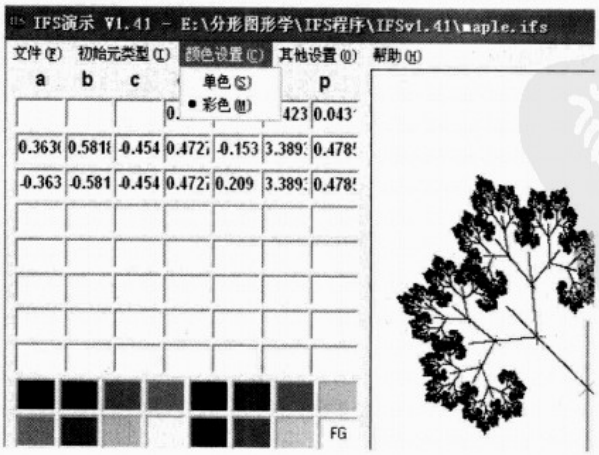


图 9.4　颜色设置

其他设置中有全屏显示和加速画图两项。加速画时，可能会出现闪烁现象，见图 9.5。



图 9.5　其他设置

当画完图形且感觉满意时，可把图画和 IFS 码都存起来。存储的位置默认为软件所在目录。其中 IFS 码文件既有 IFS 码也有 X 和 Y 的定义域。如蕨叶的 IFS 码文件是这样的：

-0.6 0.6 -0.1 1.5
0 0 0 0.16 0 0 0.01
0.2 -0.26 0.23 0.22 0 0.2 0.07
-0.15 0.28 0.26 0.24 0 0.2 0.07
0.85 0.04 -0.04 0.85 0 0.2 0.85
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

本软件可到网站 http://www.is.pku.edu.cn/~zxp/下载。

9.2　ChaosPro3.2 介绍

9.2.1　简介

ChaosPro 是一款非常优秀的分形图形制作软件，它支持多种类型的分形，具有强大的三维效果、动画效果，而且是完全免费的。本软件可从 http://www.chaospro.de 下载。

打开 ChaosPro 之后，首先看到的是一幅 Mandelbrot 分形。用户可以在右侧的面板上调整各种参数，如显示区域、角度、迭代次数、图形大小等。这些参数值被修改后，ChaosPro 会根据参数的新值立即重新绘制分形。此外，用鼠标左键双击图形可以把图形放大 2 倍，右键双击则是缩小。如果要放大图形的某个特定局部，可以使用 Fractal 菜单的 Select 菜单项。用户也可以在图形上按下鼠标后任意拖动，以调整显示范围。读者可

以尝试多次放大 Mandelbrot 集并适当拖动，观察效果。如果要改变图形的颜色，可以按下工具栏的（调色板）按钮，然后选择不同的颜色组合。工具栏上的大部分按钮含义都很明显，比如绿色交通灯按钮代表“开始绘制分形”，红色交通灯按钮代表“停止绘制”。

值得指出的是，打开右侧 Colorcycling 面板的 Color Cycling 单选框后（图 9.6），会产生非常绚丽、震撼的动画效果。这个功能适用于所有类型的分形图形。

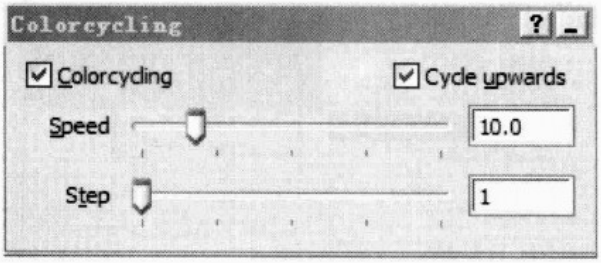


图 9.6　Colorcycling 面板

如果要绘制其他的逃逸时间法分形，可以先在右侧 Parameter 面板上点击 Formula，然后点击按钮，就可以从弹出的公式选择窗口（图 9.7）中选择其他分形如 Julia 分形等，在 Ultra Fractal 文件夹中有非常多的分形可供选择，读者可以自行尝试。要注意的是，此时只能选择逃逸时间系统类型的分形，如果选择其他类型的分形（比如 Julia Object），则 ChaosPro 会提示“Wrong Formula”，因为 Julia Object 是 Quaternion（四元数）类型的分形，而当前打开的 Mandelbrot 分形是 Escape time（逃逸时间）类型的。

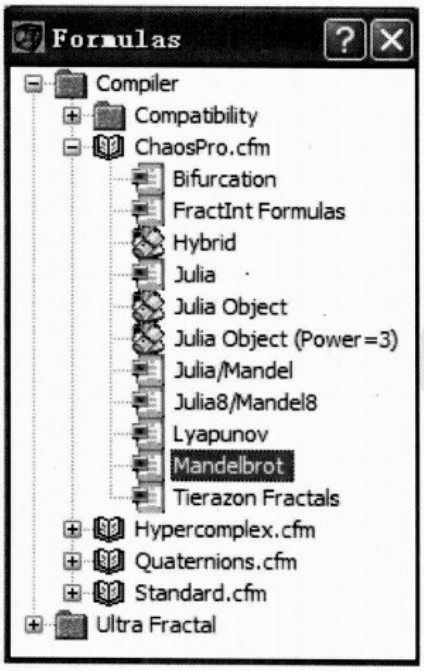


图 9.7　公式选择窗口

如果要绘制其他类型的分形，我们应首先从 Fractal 菜单中的 New Defaulttype 子菜单中（图 9.8）选择相应的分形类型，如 IFS，L-系统，动力系统（Dynamic system），四元数（Quaternion）等。

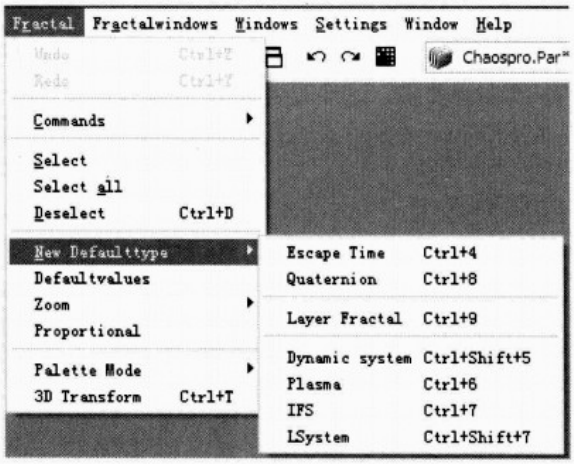


图 9.8　分形类型选择菜单

选择了某个类型的分形后，比如 IFS，我们只要按下工具栏上的绿色交通灯按钮，就可以看到 ChaosPro 为我们预定义的蕨叶图形了（图 9.9）。当然，我们也可以设置一些基本参数，比如迭代次数（Iteration），着色模式（Colormode），旋转角度等。

如果要生成其他的 IFS 分形，可以有两种选择：一种是生成预定义的分形：在主界面右侧的 Parameters 面板上点击 Formula，这里的 Formula 下拉组合框中包含了多种预定义的 IFS 公式，默认值为 fern。其他类型分形的操作与之类似，总之我们都可以在 Formula 面板中选择不同的预定义分形，读者可以自行尝试。另一种选择是自己给定 IFS 参数来创建分形，这需要用到 ChaosPro 提供的分形参数编辑器。下面，我们将详细讲解每一种类型分形的参数设置方法。

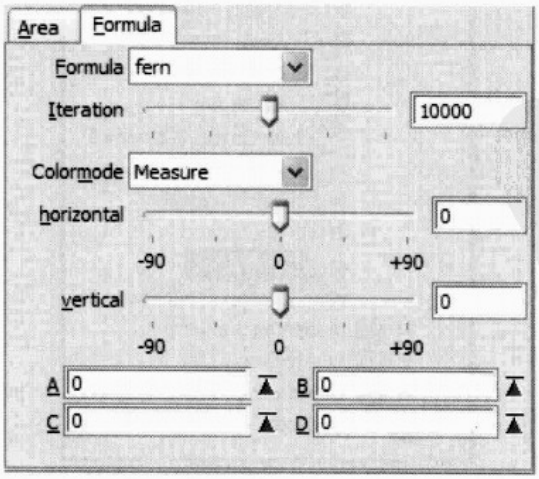


图 9.9　IFS 公式选择面板

9.2.2　各种类型分形的参数设置

1. IFS

要自定义 IFS 参数，应首先选择主窗口中 Windows 菜单的“Formula Editor IFS”，打开图 9.10 所示的 IFS 公式编辑器。此时左上方的 Formula 列出了多个预定义公式。右侧的 Actions 列出了可以进行的操作，如打开（Load）一个*.ifs 类型的公式文件，把一个公式保存（Save）为.ifs 文件，新增（Add）一个公式等。在这个窗口中进行公式编辑以后，就可以在图 9.9 所示的公式选择下拉框里选择新的公式了。

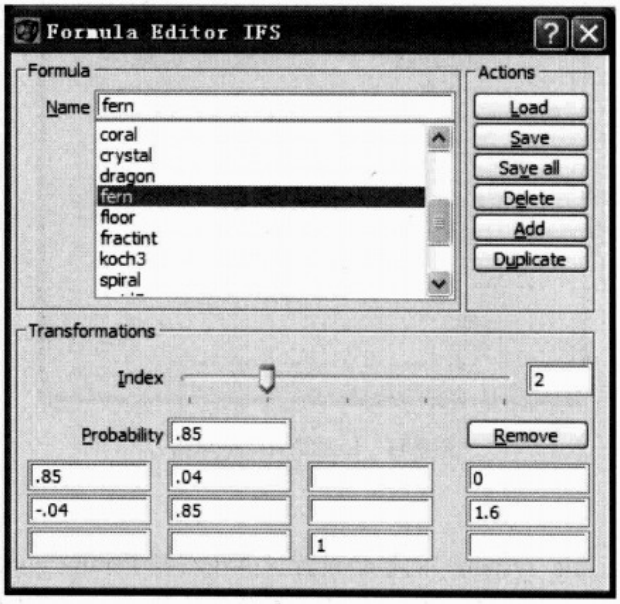


图 9.10　IFS 公式编辑器

窗口下方的 Transformations 定义了一个 IFS 变换的各项参数。我们知道，一个 IFS 由一组变换公式组成，这里的 Index 代表公式编号。比如，fern 就包含四个公式。每个公式包括一个概率值，一个 3*3 的矩阵和一个 3 维向量，也就是说，默认情况下，我们定义的都是一个 3 维立体的 IFS。ChaosPro 规定左边的 3*3 个输入框代表矩阵，右边的三个输入框代表向量，如图 9.10 所示，fern 的第二个公式概率为 0.85。

对于绘制出的 3 维图形，我们可以通过设置图 9.9 中的 horizontal 和 vertical 值来旋转观察。如果要定义二维的 IFS，参照图 9.10 所示，只要把每个公式中最下面一行的值都设为 0，0，1，0 即可。读者可以比较预定义的 fern 图形和 3d fern 图形的不同，然后尝试定义自己的 IFS 参数。

2. L-系统

与 IFS 类似，自定义 L-系统的公式时，应首先打开主窗口中 Windows 菜单的 Formula Editor LSystem，如图 9.11 所示。窗口的上半部分与图 9.10 类似，下半部分则定义了一个 L-系统公式，包含三个部分：初始元（axiom），角度（angle）和一些生成元（变换

规则）。定义这些规则后，就可以在主界面右侧的 Parameters 面板里选择公式和设置迭代次数，绘制分形了。

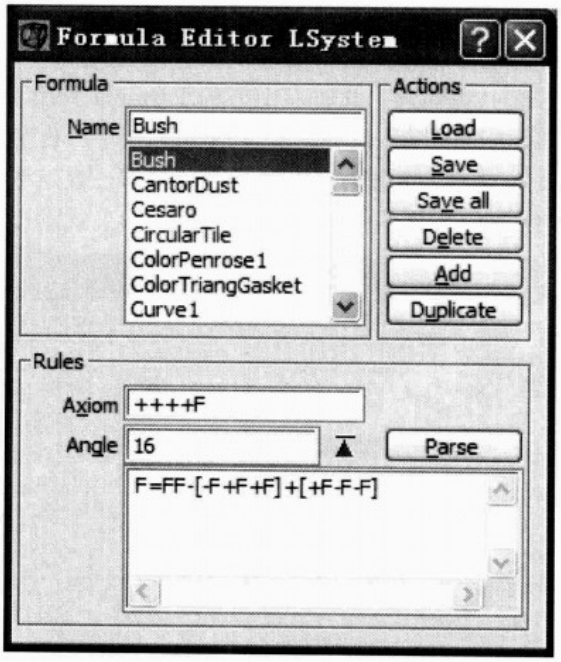


图 9.11　L-系统公式编辑器

3. 动力系统

动力系统（dynamic system）的参数设置只能在右侧 Parameters 面板内进行。通过改变 Formula 类型和 A、B、C、X、Y、Z 等参数的值，可以得到不同的图形。值得指出的是，把 Coloring（着色模式）设置为 Time、Position，再结合 Color Cycling 时会产生更好的动画效果。

4. Plasma 和 Quaternion

Plasma 分形的参数较少，这里不做解释。建议把 Color cycling 打开，观察动画效果。Quaternion 的参数虽然比较多，但是大多比较直观，如光照位置、光照强度等，这里也不再一一介绍，建议读者自己尝试。

5. Fractal Formulas 编辑器

对于除了 IFS、L-系统类型之外的其他分形，如逃逸时间法分形，四元数分形等，我们都可以通过 Windows 菜单中的 Fractal Formulas 编辑器来编辑，如图 9.12 所示。ChaosPro 为我们提供了一套类似 C++/Java 语法的分形编程语言，内置了复数、四元数等多种分形相关的数据类型，使用起来非常方便。

以 Mandelbrot 分形为例，我们在图 9.12 中可以看到四个函数定义：

void init(void)：初始化函数，在对每个点进行迭代前执行；

void loop(void)：迭代函数，这个函数定义了迭代公式；

bool bailout(void)：迭代条件函数，这个函数决定了迭代次数，每次迭代结束后计算此函数，如果返回 true 则继续迭代，否则结束迭代；

void description(void)：描述函数，定义了公式的各个参数和名称，我们看到图 9.13 下半部分的 criticalpoint，Bailout Value 输入框就是由这里生成的。

如需要了解更多的语言细节，请参考 ChaosPro 的在线帮助手册的 Language Reference 部分。

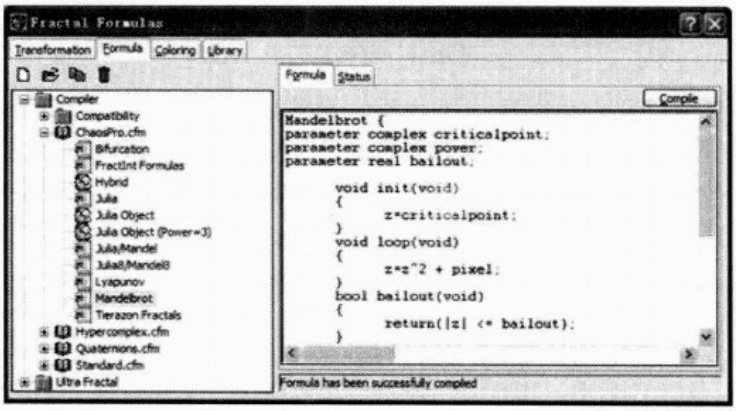


图 9.12　分形公式编辑器

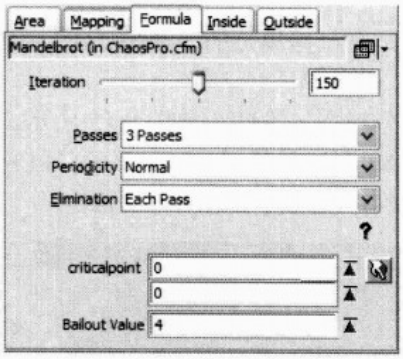


图 9.13　Mandelbrot 分形的公式参数设置

6. *层编辑*

除了根据一个给定公式绘制某一种特定类型的分形，ChaosPro 也支持“Layer”功能，即把多种分形混合显示在一张图中，达到更绚烂的艺术效果。我们不妨打开一个 ChaosPro 自带的例子。注意工具栏上有两个下拉组合框，左边的组合框代表当前打开的分形文件，默认值是 ChaosPro.par；右边的组合框里可以选择这个文件中定义的分形参数。我们一开始看到的 Mandelbrot 分形就是由 ChaosPro.par 文件中名为的 Startup 预定义参数生成的。在组合框中选择了其他的预定义参数后（比如 FirstTry），并按下绿色交通灯按钮后，就可以看到一幅非常复杂的分形图形。观察 Layer Properties 面板（图 9.14），我们可以发现这里其实定义了 4 个叠在一起的分形。

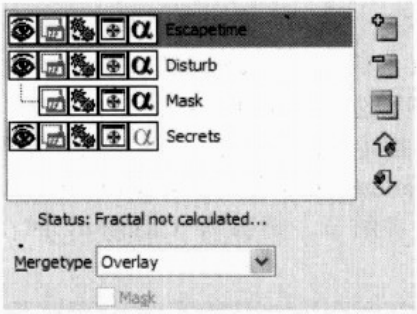


图 9.14　层编辑面板

在这个面板中，我们可以进行添加层、删除层和修改层的透明度等操作。用过 Photoshop 的读者应该会很熟悉这些操作，这里的细节我们就不一一介绍了。要创建一个多层分形图，只要选择 File-Fractal-Layer Fractal 即可，然后读者便可以发挥想像力，创造出美丽的图片。

7. 3D 图形变换

ChaosPro 支持把某些类型的分形转换到三维，包括 Julia、Mandelbrot、Plasma 和 Lyapunov Space 这几种类型，其原理是把迭代次数作为三维空间中 Z 轴的高度信息。转换方法很简单：按前面讲过的方法生成上述任一种分形后，点击主窗口中 Fractal 菜单的“3D Transformation”即可。

不过，这个转换过程还是有一些技巧的。我们以 ChaosPro 自带的“Black Hole”分形为例，说明 3D 转换中的一些细节。

默认状态下，绘制出的分形是二维的，如图 9.15 所示。点击 3D Transformation 菜单项后，我们看到的并不是漂亮的 3D 图形，而是一个黑白相间的矩形块。这是因为三维空间的高度参数设置不正确，我们需要重新设置范围。

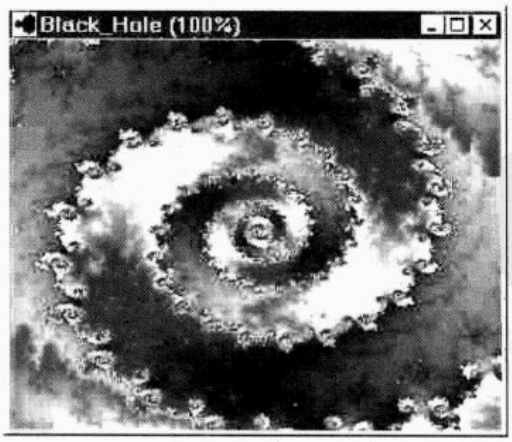


图 9.15　二维“黑洞”分形

打开主界面右侧 Fractalwindows 下的 3D Parameter 面板，如图 9.16 所示。我们可以按下“Suggest”按钮，让 ChaosPro 为我们设置一个合适的视角，然后我们就可以看到

一幅三维的“黑洞”图形了。但是，我们发现，此时生成的三维图形有非常多的锯齿，看起来很不美观。我们可以打开 Colors 选项卡，然后选中“Smooth heights”单选框，就可以平滑图形，得到更好的视觉效果。此外，在 Colors 选项卡中单击“Solid”单选框，使其不被选中，则可以产生彩色的分形。至此，我们就得到了一幅较为美观的彩色三维分形图片，如图 9.17 所示。

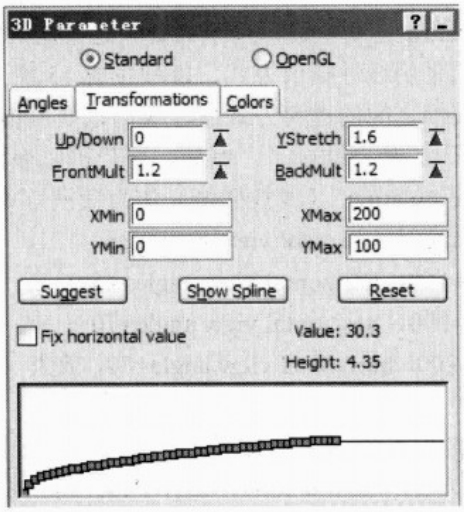


图 9.16　三维参数设置

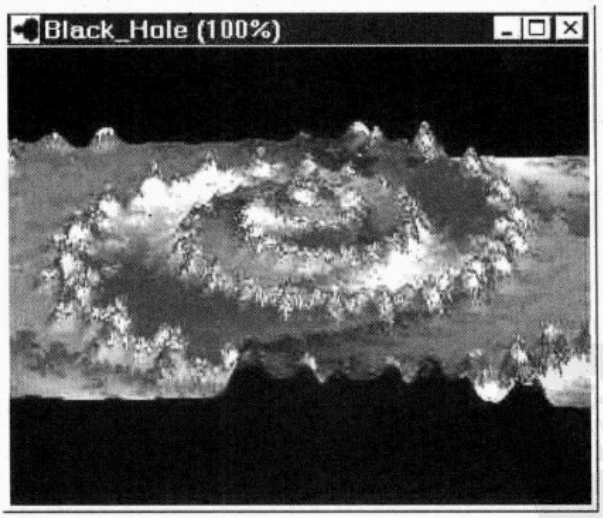


图 9.17　三维“黑洞”分形

在 3D Parameter 面板中还包含很多其他参数，读者可以自行尝试修改这些参数，观察所得到的结果。比如，Angles 选项卡包括了观察角度和光线强度相关的一些设置。此外，默认情况下 ChaosPro 采用“Standard”模式生成图形，如果改为“OpenGL”模式，则生成的图形质量可以得到进一步提升，但相应代价是生成速度大大减慢，并且要求非常多的内存。

8. 制作动画

ChaosPro 可以生成 AVI 格式的动画。基本过程如下：

用户首先向动画系统添加一个初始分形。然后，用户对该初始分形任意作一些修改，比如修改迭代次数、旋转角度等，并把修改后的分形再次添加到动画系统中。多次重复这一过程，动画系统中就包含了一系列基本分形图形，我们称之为“关键帧”(Key Frames)。我们希望 ChaosPro 自动生成每两个关键帧之间的过渡动画。为此，用户还需设置过渡方式的细节，如采用线性插值方式，每秒生成 15 帧图片，是否为 3D 动画等。

设置好关键帧及关键帧之间的过渡方式后，我们只要再指定动画的分辨率和格式，就可以让 ChaosPro 自动生成分形动画了。例如，用户可以根据以下参数添加 4 个关键帧，并选择了“linear interpolation”（线性插值）作为变换方式：

bailout=60，maxiter=80，horizontal view angle=30

bailout=80，maxiter=160，horizontal view angle=45

bailout=80，maxiter=200，horizontal view angle=70

bailout=70，maxiter=200，horizontal view angle=70，放大一倍（双击窗口中心位置）。

详细操作步骤如下：

1）任意绘制一幅分形图形作为初始分形。

2）点击 Windows 菜单的“Animation”，选择“Key Frame Path”选项卡，然后点击“Add First”按钮，即可把上一步绘制的初始分形加入到关键帧列表中。

3）然后可以修改原先绘制的初始分形，最好不要作过多修改，否则生成的图形看起来可能会很奇怪。

4）把修改后的分形加入关键帧列表。方法是先点击选中前面加入的第一幅关键帧，再点击“Add”按钮，这样就可以把新加入的关键帧放在第一幅关键帧的后面。

5）重复 3、4 两步，添加更多的关键帧。

6）完成关键帧的添加后，在“Settings”选项卡中设置动画分辨率（高度和宽度），每秒生成的帧数，插值模式，是否为 3D 图形等。

7）最后切换回"Key Frame Path"选项卡，按下“Start”按钮，就可以开始生成动画了。

我们可以发挥想像力以创作出各种各样的动画。比如先绘制一幅 Julia 分形，适当设置 3D Transform 参数，这样就得到了一座“分形山峰”，然后“拉近镜头”（取消 3D Transform，在二维的分形上双击，使之放大），向前“飞行”一段（在窗口中拖动分形），并且在飞行的同时看到色彩变幻（修改调色板），分形山峰忽高忽低变换不停（修改 Julia 分形的参数 C 的值）。最后，光线变暗（修改 light 参数），一切湮没在黑暗之中……

通过网络下载最新的分形公式，我们可以不断充实我们的分形图形库。在主界面右侧的 Update Formulas 面板中，我们只要点击“Start”按钮，然后等待下载即可。

9.3　综合绘图软件 fe

fe（fractal explorer）是乌克兰的分形学家研制成功且基于 Windows 平台的分形综合绘图软件。

fe 是一个可以自由获取的分形发生器，读者可以从下列网站下载：

http://www.eclectasy.com/Fractal-Explorer/index.html

fe 可以由分形数学公式产生各种各样的绚丽多姿的分形图像。它既能画出传统的多项式分形如 Mandelbrot 集、Julia 集和 Newton 集等，也能画出迭代函数 IFS 和奇异吸引子的图像，还能产生 4D 的四元数分形。该软件有许多独特的画图功能，对同一幅图像，能变幻出多种色彩的画面效果。

fe 是从 1999 年开始发展的，从公开推出的 1.04 版，直到 2004 年 7 月的 2.01 版和 2005 年 1 月的 2.02 版。每个版本都会有新的创意。图 9.18 是 2.01 版和 2.02 版的主界面。



图 9.18　2.01 版主界面

FE 最突出的功能是多项式分形和四元数分形。点击界面最左边的曼德勃罗集图案，再点击同一排的$\sqrt{\alpha}$，就见到图 9.19，其多项式内容含

1）曼德勃罗型分形（Mandelbrot-style fractals）；

2）牛顿型分形（Newton-like fractals）；

3）超牛顿分形（Super Newton fractals）；

4）作者们自创的分形（The author's fractals）；

5）特洛茨基公式（J. Trotsky's formulas）；

6）像凤凰的分形（Phoenix-like fractals）；

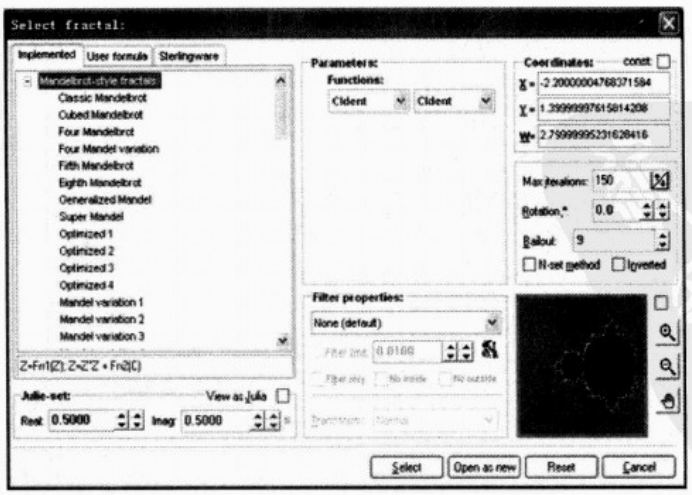


图 9.19　曼德勃罗型分形的分类

7）非线性公式（Non-linear formulas）；

8）正弦-余弦公式（Sine-cosine formulas）；

9）分形轨道（fractal orbits）。

每一类都包含多种公式，如曼德勃罗型分形，就多达 27 种公式类，见图 9.19。为了更清楚对图 9.19 进行说明，可见图 9.20。

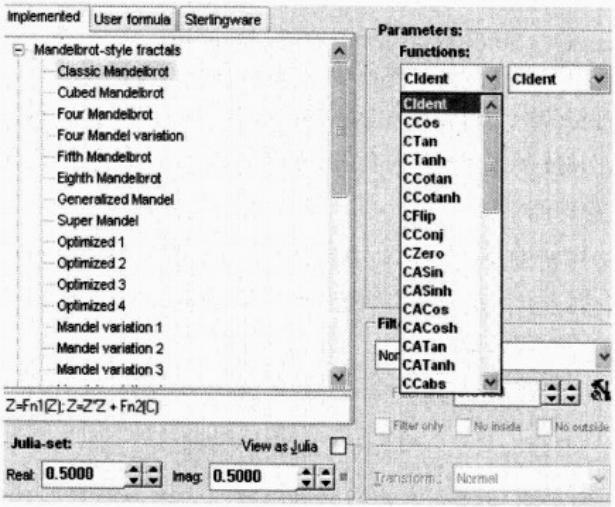


图 9.20　图 9.19 的部分图

图 9.20 是经典(Classic)曼德勃罗型分形画图界面，其迭代公式是 Z=Fn1(Z); Z=Z*Z+Fn2(C)。右边的两个函数分别对应 Fn1 和 Fn2 且都是复变量函数。其中 CIdent 表示 Fn1(Z)=Z，Fn2(C)=C。选择不同的 Fn1 和 Fn2 就得到不同的（广义）曼德勃罗分形。

图 9.20 中有 View as Julia 指示，若在该方框中打勾，出现的画面就是茹莉亚分形，见图 9.21。

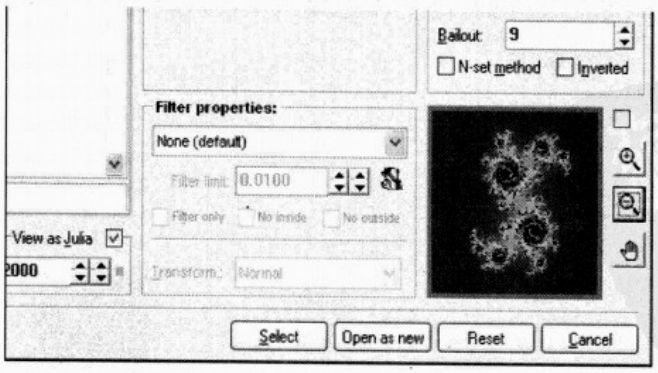


图 9.21　Julia 集

待所画图形觉得满意后，可选择 Select 框，退出小画面进入主界面，这时可从 Size 框中选择所需要的尺寸就行了。

fe 的另一个特色就是四元数分形。

在主界面上点击 Q 字，就可见到一幅四元数茹利亚集，在点击 $\sqrt{\alpha}$ 后，得到如图 9.22 所示界面。

使用者可调节四元数的四个参数 C.real，C.imag.I.，C.imag.j.和 C.imag.k，且同时可见右面画框中图形的相应变化。当认为满意时，可再选择颜色。步骤是：右下角的黑圆表明背景色是黑色，如想换成其他颜色可点击它得到人们通用的图 9.23，在该图上可选择任意颜色作为背景色。

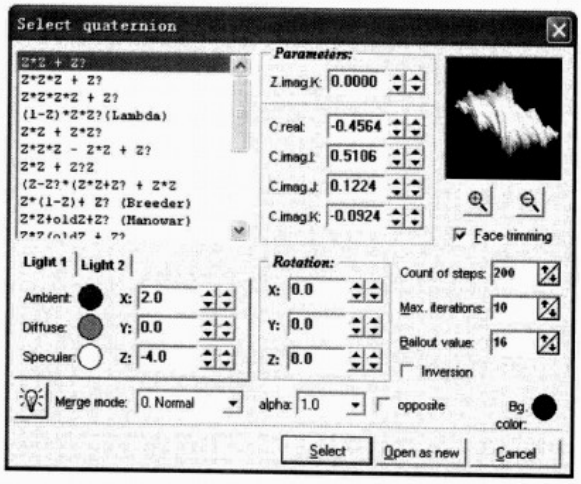


图 9.22　绘画四元数儒利亚分形的界面

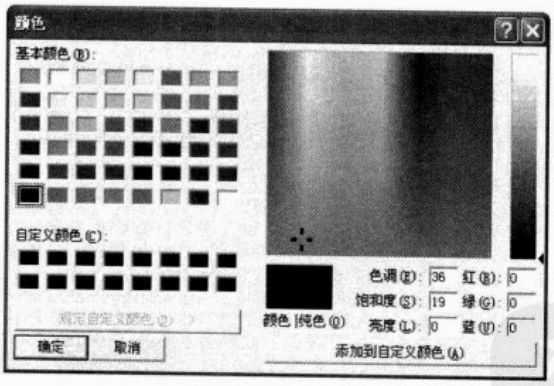


图 9.23　选择背景色

选定背景色后，还能选择三个坐标轴（X，Y，Z）的旋转（rotation）角度，还能通过调节 Ambient、Diffuse 和 Specular 的三个颜色去改换图案的颜色，还能通过选择界面上的 Merge mode，使得图形更好看。

在图 9.22 的左上方，可进行茹利亚集的函数选择，从而得到更多的广义四元数茹利亚分形。

在小图上画好图形后可点击 Select，图形就显示到主界面了。

注册版的 fe 也能画山水风景，其部分作品可在光盘中找到。

9.4　分形山地风景软件 Terragen

已经有若干可以自由获取的分形山地风景软件了，如 M3d，Genesis 等。这里介绍的是由英国学者研制成功的 Terragen。

Terragen 是一个风景编辑的软件，虽然正在发展之中，但是它却已经能够提供令业余和专业人士都很满意的画面质量了(见图 9.36～图 9.38)。读者可以到下列网站下载：

http://www.planetside.co.uk/terragen/

新的版本是 v0.9.43. 是 2005 年 9 月完成的。

Terragen 工作的基本原理如下：

1）创建或者输出一个黑白图来表征每个点的高度。这幅图就被定义为一个高度的分布。

2）定义每一部分的地形要采用什么颜色，以区别曲面的坡度和高度。

3）调整大气的参数来表现雾或者日落等。

4）调整太阳的位置参数以改变整个图片的照明效果。

5）在图片中创建出云彩。

6）最后在高分辨率下输出图样，随着场景参数的变化，这个过程可能消耗几分钟，也可能耗费更多时间。

下面简介 Terragen 的用法及其背景知识。

9.4.1　主界面介绍

打开 Terragen 后可以看到左侧有一排图标，见图 9.24。

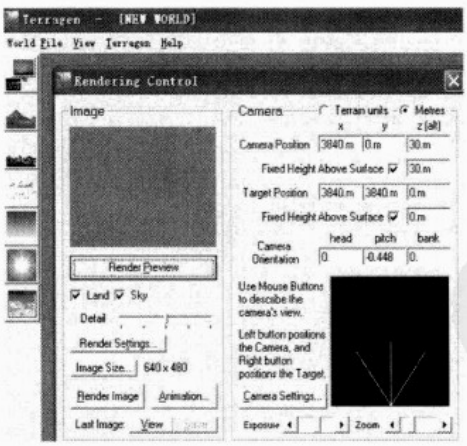


图 9.24　Terragen 主界面之部分图

点击不同的图标会弹出相应的对话框，自上而下有执行控制（Rendering Control）、地形景色（Landscape）、水面（Water）、云彩（Cloudscape）、大气层（Atmosphere）、光照状态（Lighting Conditions）和最后执行图片（Last rendered Image）等窗口。第一个图标对应的 Rendering Control（执行控制）对话框是主窗口，其中的 Render Preview（执

行预览））按钮非常有用。每当我们在某个对话框中对地形参数做了一些调整后，就可以单击该按钮快速预览生成的地形。

Rendering Control 对话框用来预览场景、调整参数并得到图片。按下 Render Preview 键，立即就会得到一幅模糊的图，上面可以看到水平线和一些隐约的云彩。除了云彩的缺陷，整体效果还是不错的。修复的方法很简单：在 Render Preview 键下面的细节（detail）滑条控制着在预览中细节的表现程度，当滑条在中间时是预览的极佳的位置。当你需要观察到更细节的信息时，将滑条右移动并再次点击 Render Preview 就可以了。我们可以将滑条放在最右面，这时候需要稍微长的时间等待，但是更多的细节带来的效果却是非常的好，所以预览的时候放在第三或者第四的位置就足够了。但是最后生成图片时，一定要放在最右边。

点击细节滑条下面的设置，会弹出一个有很多选项的窗口。首先是 Quality，它有两个滑条，分别表征着大气精度和云彩精度。默认值在大多数图片的情形中已经足够好了，但是如果最后生成图片的大气或者云彩里面产生了较严重的条纹，就需要提高大气精度。需要指出的是，提高这些参数会很大程度上增加运行时间，所以需要慎重调整。

在这之下是反锯齿处理（Anti-Aliasing）的参数。混淆现象是确定解的副产物，在画面上，垂直或者水平的线看起来是很完美的，但是其他的就会看起来参差不齐。为什么会如此呢？简单地说，显示器的显示是尽量详尽的，由于屏幕是由叫做像素的方形区域组成的，曲线和斜线很难在屏幕上真实地表现出来。解决这个问题的方法就是反锯齿处理，这也是计算机图形学的一个很大分支。Terragen 采用的方法是快速亚像素平滑方法，它的原理就是自动地在更高的分辨率下产生图像，然后重新排列这个图片，从而得到更好的细节效果。事实上，我们做的就是在每个像素里面得到更多的细节信息。

现在点击窗口上面的选项键（Options），这时候会弹出一些选项框以及我们已经很熟悉的亮度校正。描写地形和描写天空/云彩键仅仅决定了 Terragen 最后是否在图片中生成这两部分的效果。大多数时候，这两部分都是必要的，但是有些时候，当你调节画面的某些更细问题的时候，你可能仅仅需要其中一个。需要引起注意的是，这两个选项可以直接在 Rendering Control 对话框找到，所以这里它们并不重要。

下一项选择，即后面地形选择，这是大部分时候用不到的。

点击高级（Advanced）选项，你可以看到一项内存使用的输入框。它会提示说需要三分之一左右的内存来实现最优化，但是其实这个参数对绘图时间的影响并不是很清楚。如果你的内存是 64M 的，请输入 21，如果你的内存是 256M 的，请输入 64 等。

现在可以关掉这个窗口，并回到 Rendering Control 窗口。在 Detail 滑条的下面，有两个选择框分别标明为大地（land）和天空（sky）。这和选项窗口中的描画地形和描画天空/云彩键是一样的，在这里仅为了方便用户使用。

下面是图像尺寸的选项，用户可以根据最后的用途来更改。需要记住的是，增加图片的尺寸同样会增加最后的绘图时间。

右边是摄像（camera）控制，稍后会更详细地介绍。首先，我们先介绍曝光（exposure）和放大（zoom）控制。曝光是在图片上增加或者减少光线的一种重要的控制选项，在现实生活中，当外面阳光充足的时候，并不是所有的东西都是很亮的，这是因为我们的眼

睛可以看到很大范围内的色调，但其中一部分在电脑上并不能表现出来。为了让电脑上的图像看起来更真实，我们需要采用一些算法来控制图片的亮度，把色调范围调整到可控制的范围之内，曝光功能就是允许你控制这部分。在低光条件下，这个选项非常的有用，比如日落的时候。把曝光向右或者向左调一下，看看会有什么效果？接下来是放大功能。这个功能是用来控制你想显示画面为多大。当你试着移动这个滑条的时候，你会发现图片中的白色的线也会发生变化。这是因为三条白线的起始点就是相机的位置。最外面的两条线显示的是你视角的极限，中间的那条是连接相机和目标物的。如果你左移，它会自动使相机移动到那一点，如果你右移，就相当于移动了目标物。

点击相机设置键（camera settings）会出现一个窗口，上面有我们刚刚讨论过的曝光和放大控制，同时还有图片要放的胶片的设置。一般都是使用默认值，当然你也可以试改变一下会出现什么效果。稍后我们会重新讨论相机的设置。

9.4.2　地表景色

点击屏幕左边的有着绿色山脉的图标，地形对话框就会出现。这时候该窗口看起来可能使人不知所措，一个大的黑框和一些小的输入框。这个大的图片是用来描述山地高度分布的。现在这个版本里面，Terragen 只支持方形的区域，当然也可以改变某边上的像素。地形的默认值是 257 像素，这对刚刚开始运用者已经足够了。在 Terragen 里面创建高度分布，有几种不同的方法。首先，可以利用画图功能来画出山脉和峡谷，即在地形对话框，点击推土机的图标或者绿地的图标。如果左键点击黑色的窗口，就会发现已经创建了一些浅灰色的白点。利用左键点击将会提升地形，比如创建山脉。试试左边那三个并列的图标，就会发现它们其实功能一样，只是创建的规模有所不同。现在已经得到了充满不同线条和白色水滴的屏幕，这时候试着右键点击，就会发现右键点击可以创建很多深色的点，而且如果坚持时间久一点，就会变成黑色的点。右键点击降低地形，比如创建峡谷等。可以发现，如果降低地形足够多，屏幕的一部分会变蓝，这意味着已经把地形降到了水平线以下，Terragen 显示这时候地形已经有一部分没在水中了。下面是一个运用以上方法绘制的地形图，见图 9.25。

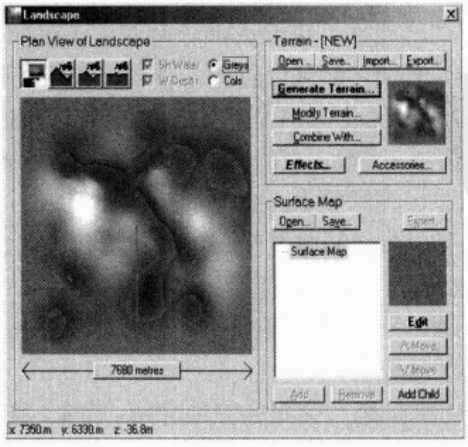


图 9.25　地形图

这个时候到 Render Rreview 对话框，选择生成预览的话，看到的就不是平面地形图，而是效果不错的立体山脉，像下面图 9.26 那样。

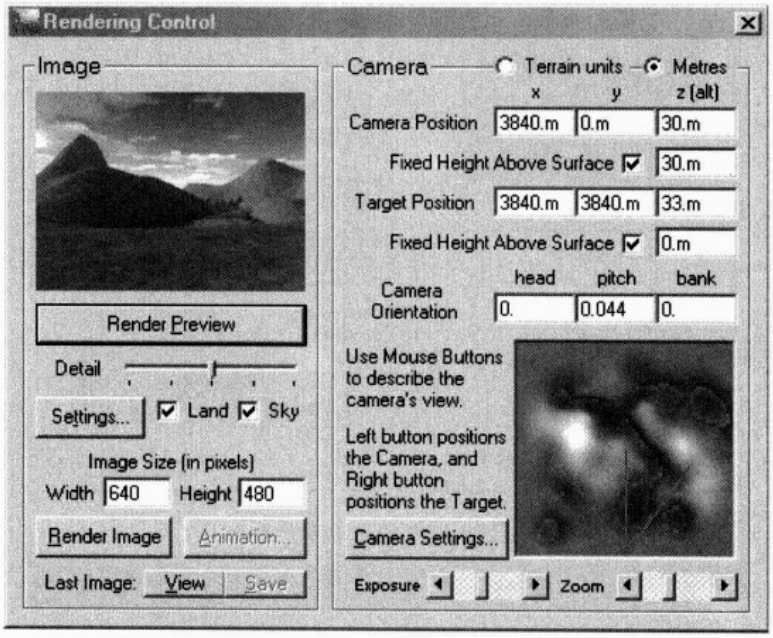


图 9.26　山地预览

显然，这比平面图有趣多了，但是看起来还并不够真实。问题是这个高度视域还是太普通了。实际的地形有很多微妙的变化和很多突然的升降，但是在刚才的地形中都没能反映出来。为了达到那种真实效果，必须在其他地方找到地形表现的资源。幸好，Terragen 有一些更好的地形生成方法，点击“产生地形”（Generate Terrain）键，如果点击 Subdivide & Displace II，就会看见一个有着四种算法的用来创建高度分布的下拉菜单。人们可以试试每一种，只要点击它并点击 Generate Terrain 就可以看到所创建且快速显示的效果。当创建内部的地形时，通常选择 Ridged Multi-Perlin，因为它的效果实在是逼真。右边的四个滑条每个都以不同的方式控制了地形的生成，想了解它们的运作，最好就是试验一下。Size of Features 滑条会很大程度上影响地形的效果，所以这是最先要试验的选项。在反复试验之后，回到 Rendering Control 对话框，并且生成预览看看效果。一旦得到了满意的效果，就可以关闭窗口了。

除了在 Terragen 里面建立一个地形文件外，还可以打开一个已存的地形文件（.ter）或者导入一个 RAW 图像文件（.raw）。在 Terragen 中，文件导入和导出还是比较繁琐的，能够帮助做到熟练处理地形文件的许多插件并不简单，所以在这里先不予讲述，还是回到处理 Terragen 自己生成的地形文件上来。在 Rendering Control 对话框里，用鼠标可以将摄像头随意移动，从而可以寻找到一个最佳的观察位置和生成满意的图像，见图 9.27 和图 9.28。

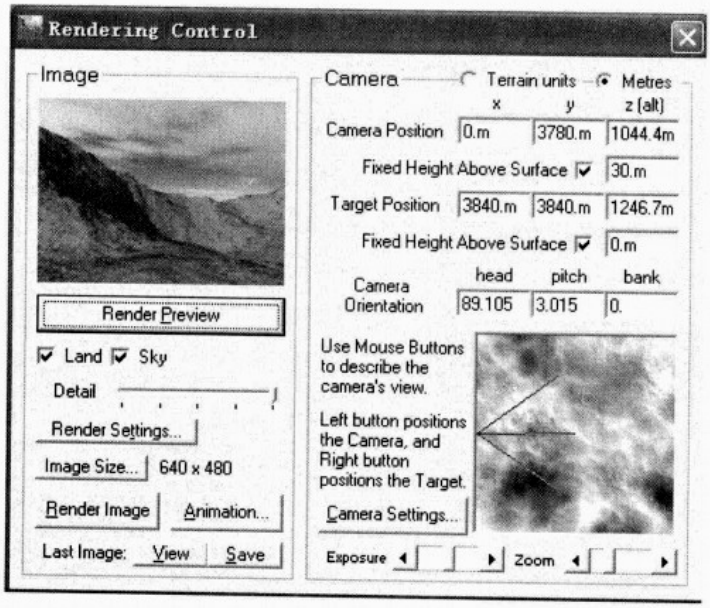


图 9.27　箭头朝向与图形之一

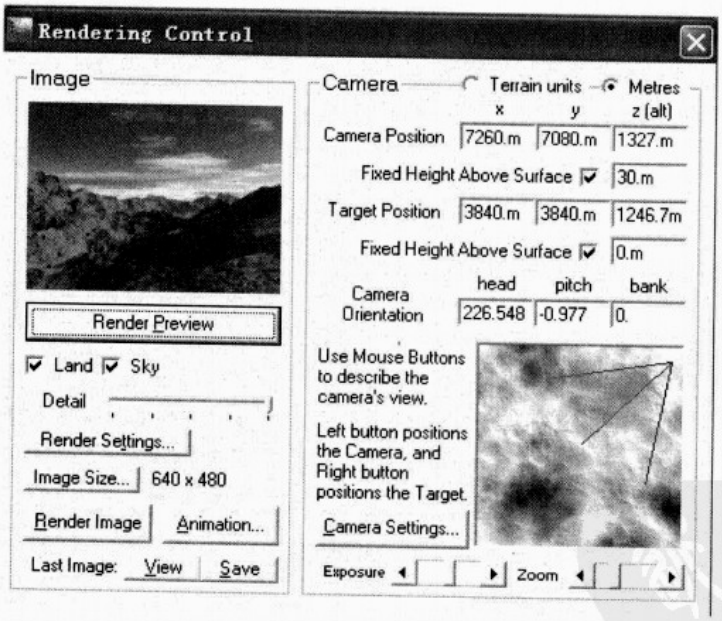


图 9.28　箭头朝向与图形之二

下面有一些注意事项：

1）尽量将你的摄像头从高处指向低处。

2）摄像头周围应该留出一点空间，这样你才可以浏览不同的地貌。

3）如果辨认地形特征有困难的话，多尝试一下通过 Detail 滑条功能创建预览。

4）可以试着随意移动你的摄像头和目标，有的时候，显现地表形状会变得比较困难，这样做可能会带来意想不到的效果。

5）尽量不要将你的摄像头放在“物像的边缘”。

回到地形景色（landscape）对话框，来看一看右下角的貌似奇怪的选项，那是唯一输入表面地图的地方。表面地图（surface map）为 Terragen 对地形着色提供了依据。比如可以处处着色为草绿色这使得山地显得很平坦。控制表面的两个重要的参数是高度和坡度，如果正确使用这两个参数的话，将会得到十分逼真的地形效果。表面地图利用的是树式的工作原理，在地形内的每一点，表面地图都从树干开始，接着逐一的走遍所有的树枝。如果那个点的特性与树枝相符合，就确定地走到这个树枝，然后再利用同样的方式逐一的检查这个树枝的子树枝，并最终确定在地形上某确定点所应该采用的颜色。制作地形中，作者采用了相对简单的表面地图，包括草地、岩石和积雪。首先可以点击表面地图，然后点击 Add Child 键四次，这样就给主表面地图加入了四个子关系。这里，一个重要的事实是：当 Terragen 制作表面地图的时候，它从顶开始，从第一个子关系开始检查，如果第一个子关系比较具有草地的特性，那么即便后面的会更具有草地的特性，系统也会选择前面的。这种情形下，这种特殊性质的草应该是普通形式的草的子关系。现在，可以使我们的表面地图运作起来了，点击四个输入中的第一个“新表面”，然后点击编辑（edit）键，这时会打开另一个窗口，在窗口的顶端我们可以输入并且重命名我们的表面。我们将此重命名为“基本草地”。左边的环形斑纹就是表面现在的颜色以及其崎岖的程度。点击颜色（colour）键，一个可以让我们改变表面颜色的窗口就会出现。如果把这里设置成深绿色，有两种办法，一种是点击右下角的绿色键，另一种是手动红，绿，蓝滑条的设置。关掉颜色选择的选项，就可以看到左边的颜色环形斑纹已经变成了深绿色。除了这个斑纹，还有两个滑条可以用来控制地形的崎岖度。由于草地的表面是崎岖不平的，我们需要把崎岖度（bumpiness）滑条一直调到最右边。模拟地形（mimic terrain）滑条控制着崎岖度和下面的地形相比的相似度。点击窗口下半部分的高级分布键（advanced distribution），在新出现的窗口里，有很多的复选框，滑条以及其他需要掌握的控制功能。首先是左边的小框，这个框大致表明了表面会放在地形的哪个部分，白色的区域就是表面的区域，黑色的恰恰相反。右边的覆盖滑条控制的是整个物像中被表面覆盖的比率，建议这个参数到最后去做出调整。分形噪声（fractal noise）设置控制着表面的复杂度，在草地中，可以将这个参数设置得比较中等，但是在大面积的表面，比如雪地中，这个参数就要设置得低一些。

接下来是高度约束设置。一般地总希望草地生长在各个高度，所以这里没有给草地的生长高度设置任何的约束。但给草地生长区域的坡度做了限制。在坡度约束设置里面，把草地生长的区域的坡度限制为二分之一，所以需要相应的移动滑条。注意到，在移动滑条的同时，方形的表面地图也会根据设置相应地做出调整。由于不想给草地生长的区域的坡度作出最低的限制，所以对其没有做出选择。在这些滑条的右边是控制表面边界的，可以使边界十分的清晰，也可以很模糊，也可以居中。在前面的草地情形里，可使用默认值，而事实上，默认值的效果还不错。既然已经给出了草地区域的所有的限制，现在需要设置的是一共需要多少草。把覆盖滑条移动到某个恰当的位置，使得预览窗口出现适量的绿色分布，见图 9.29。

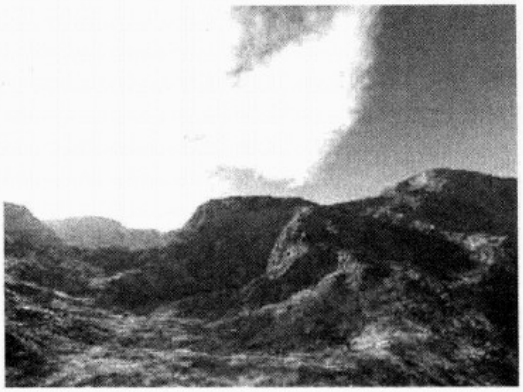


图 9.29　有绿色草地的预览图

回到地形景色窗口并且点击表面地图的第二个子关系（child），就是剩下三个“新表面”的第一个。点击 Edit，把这个表面设置成为褐色，以表征沙地或者岩石。这些表面无需有太多的修饰，仅仅需要把崎岖度调低一点，还需要把覆盖度降低一些，并把分形噪声升高一点。同样地，再次回到 Landscape 窗口，点击第一个剩余的“新表面”。重复上面的步骤，点击 Edit，再点击颜色键，采用现有的浅褐色，并把红，绿以及蓝颜色各自调向左一点，使整体变暗。这时三种颜色的值分别为 20，18 和 16。关掉颜色选项，跳到高级分布选项，希望深色岩石出现在坡度很大的地方，比如近乎垂直的悬崖。所以选择最大的坡度，并把最小坡度调到离右端四分之一处。把模糊度调到最低，把覆盖度调大一点，来增加岩石的总量。最终的深色岩石设置参见图 9.30。

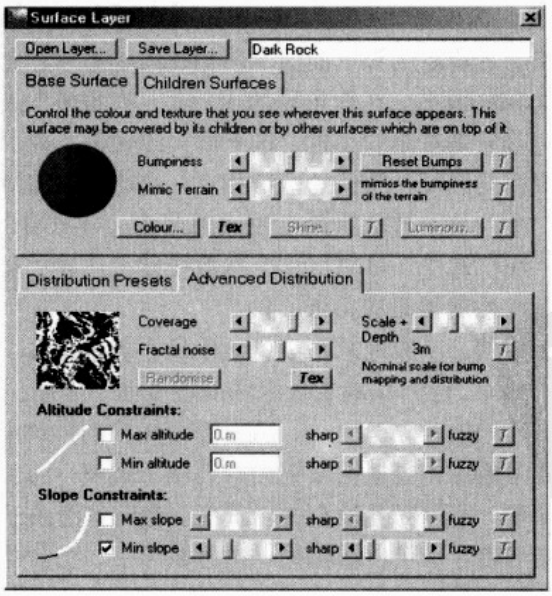


图 9.30　设置岩石为褐色

最后，回到 Landscape 窗口，并且选择最后的 New Surface。点击编辑按钮并且将这个新表层命名为 Snow。点击 Colour 按钮，将这个表层设置成纯白色。关闭 Colour Picker

并且将崎岖度设置为最低。为了将这个表层放在山顶，所以需要对表层的海拔高度做一些约束。首先得确定山的海拔高度，怎么做呢？其实，这并不难，看执行预览（render preview），会看到一个环形山脉在图像的远端。点击 Landscape 窗口，会看到摄像头和目标在地形中的具体位置。事实上，这些山脉是一些环绕在目标远端的白色的条带，如果将指针移动到这些条带上，会看到三个出现在 Landscape 窗口下半部分的指示器显示了变化的 *X*、*Y*、*Z* 值，即指针所在位置地形的坐标值。下面所需做的就是将 Z 值（海拔高度）记录下来。而在所创建的地表中，海拔高度大约是 1000 米。再回到创建的名为 Snow 的表层窗口，复选 Min altitude（最低海拔）按钮，将其设置为 900 米。注意到应将最低海拔值设置得比山的海拔高度（1000 米）要低一点，这样做是要保证能够在想要的地方得到一些雪。此外，还要将模糊度差不多调到最低，在“Slope Constraints（倾斜约束）”中，选择默认值，在模糊度选项中也将其调到了最低，这样，山脉上岩石和雪的过渡就显得非常明显了。若干预览图见图 9.31，其中右上角的图中就有雪景存在。

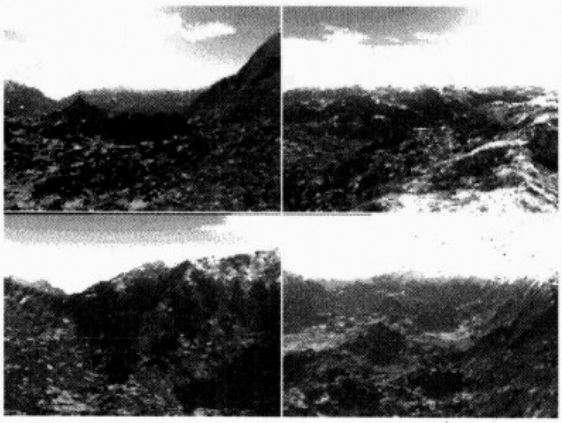


图 9.31　若干预览图

9.4.3　水面、云彩和光照

在 Terragen 中，画“水”是很简单的事。只要确定一个海平面，则位于该平面以下

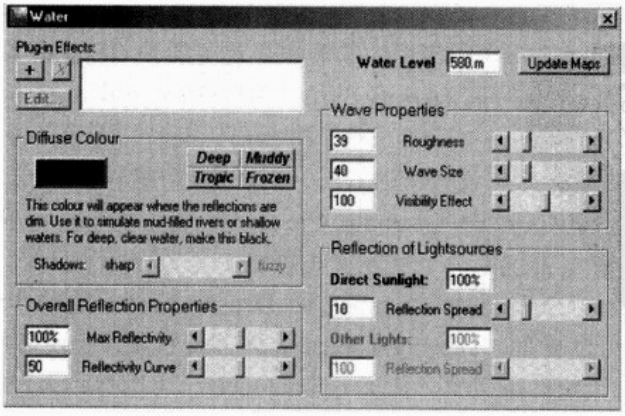


图 9.32　“水”的界面

的所有画面都成了水面。当然对于瀑布或阶梯式的湖泊是不能这种方法的，但它确能仿真自然界的河流和湖泊。对着 Render Preview 中的画面，什么地方可以成为海平面呢？只要打开 Landscape 窗口，就可以用山地的一个 z 值确定其为海平面了。这跟确定雪的高度是相似的。比如说选择 580 米为海平面，打开 Water 窗口后通过点击屏幕左边蓝色波状的图案，见图 9.32，并在 Water level 方框里填入 580 米数字，再点击 Update Maps 按钮，则在 Lanscape 窗口的山地图上就出现新的海平面了。一旦做完上述步骤，就能在 Landscape 窗口中看见山中的湖泊有多大了。如果不满意，则重新写入新的数字，再点击 Update Map，反复几次就可完成，见图 9.33。

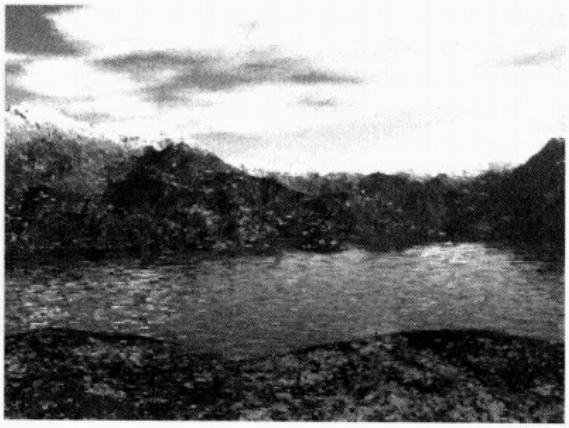


图 9.33　有湖泊的山地风景

在水平面（water level）窗口下面有四个可控制的参数，它们是：Waves（波），Reflections（倒影），Subsurface（子表面）和 Shore（滨岸）。每个选框都能分出更多的子框供调节。这里不再一一说明。

云彩的设计是 Terragen 中最简单的一部分。插入不同的数字总会留下美丽的云层画面。当然某些结果可能会好于其他的表现。

云彩设计窗口见图 9.34。

看到的界面中有许多还未显示出来而呈灰色的按钮，如 Open、Save 等。这表示这些功能还未执行，可以忽略。在能应用的选项中最常用的是 Generate Clouds（产生云彩）、Density Contrast（密度对比）和 Density Shift（密度变换）。点击 Generate Clouds，会打开其他一些窗口如 Persistence（持续）滑条和 Size（规模）滑条。Persistence 滑条控制云彩形状，较低的设置会产生较光滑的云层，反之会得到较复杂的云朵。Size 滑条控制云彩的大小，但要记住去点击 Generate Clouds 以产生基于新设置的新云朵并回到 Rendering Control 窗口去预览。一旦发现在预览中有较理想的云彩，就结束 Generate Clouds 对话且再次看一下 Density Contrast 和 Density Shift，这两个选项能用于云朵更加细腻以便得到最好的画面。Density Contrast 调节云彩和天空两者的对比度，Density Shift 则是调整云对天空的覆盖率。按下 Update view，可以得到大尺寸的预览。在上述选项的上面有几个技术性的选项。选择 3D 能使 Terragen 绘制出三维云彩，这时，Terragen 用了与普通 2D 云彩不同的算法。这种算法并非必要的，因为这会用去更长的

时间，但毕竟这会画出更出色的云朵，因而值得试一试。实用中可以把 2D 和 3D 两种算法都试用几次并用预览以得到最佳效果。

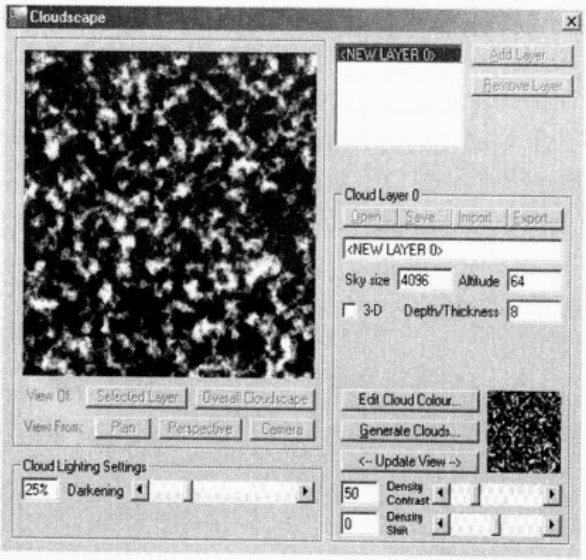


图 9.34 “云彩”界面

云彩设计的过程中，还有一些其他界面窗口，如 Sky Size，Altitude，Depth/ Thickness 和 Darkening。使用者可通过实践去体会其创作意图，这里不再叙述。

如果点击主界面左边那个像太阳的图标，就进入了 Lighting Condition(光照状态)。该窗口有两部分组成，上半部含有最重要的光照命令，下半部分有四个组件能给出更多的控制，见图 9.35。

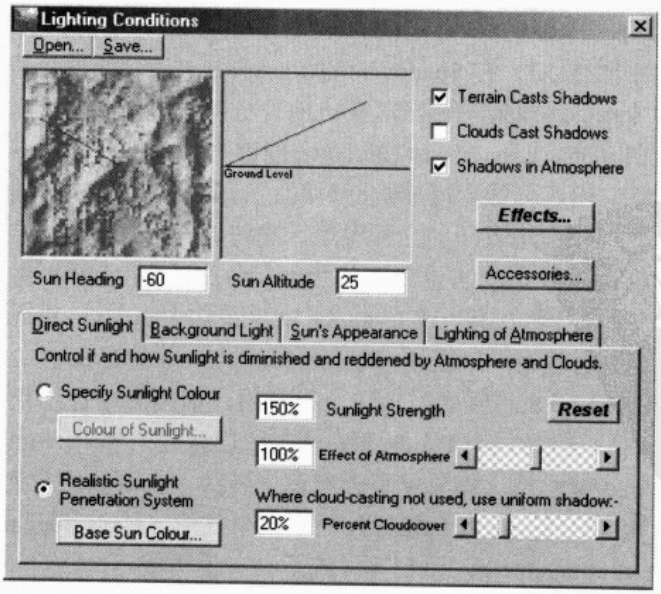


图 9.35　光照状态界面

图 9.35 上部左面的方框里显示了一个山地景色的阴影凸纹图(如第二章所言是伪三维图)，其中之线段指向山地的中心，它表示太阳从该方向照耀着这个山地。通过点击围绕山地的各个部位，可以改变太阳光的指向。通常是当太阳光指向摄像机的观察点时可获得最佳的阴影。如果希望太阳出现在画面里，则可将太阳坐落在照相机的反方向。右面这个图是控制太阳的高度，高的高度会使景色不讨人喜欢，因为缺乏对比度。因此最好把太阳高度设置在中等位置。当把太阳高度减少到落在地面上时，景色会变得黑暗，当然这时可在 Rendering Control 窗口里增加曝光度改善画面质量。应用上述两种控制，可使太阳从任意方向射来，同时当移动太阳光方向时也在图上移动了阴影。这对寻找最佳太阳位置是很有利的。图 9.36 就是上述过程中得到的六个图形。



图 9.36　不同太阳位置的山地风景

图 9.35 右边还有三个校正框，它们是：Terrain Casts Shadows(山地阴影投射)，Clouds cast Shadows（云层阴影投射）和 Shadows in Atmosphere（大气层中阴影）。Terrain casts Shadows 用于告诉 Terragen 投射阴影是落在山地自身呢还是落在水面上，这个设置必须始终坚持留用，见图 9.36。Clouds cast Shadows 能够产生某些有趣的效果，比如图中有从云层中射来的光线。但这可能会增加执行时间。Shadows in Atmosphere 允许大气层与山地相互作用，从而产生围绕太阳的光晕现象，建议始终留驻该设置，见图 9.36。

图 9.35 的下边是调节光照的更特殊的控制。在其左边有两个选择圈点：Specify Sunlight Colour（制定太阳光色调）或 Realistic Sunlight Penetration System（逼真的太阳光穿透系统）。建议将 Realistic Sunlight Penetration System 始终保留。Sunlight Strength（太阳光强度）是控制太阳光亮度的，该选项很有用。但当想用该选项时，也可考虑用 exposure（曝光）取代。Effect of Atmosphere（大气层效应）和 Percent Cloudcover（云层覆盖率）一般用缺省值就行。如果觉得云彩过于怪异，也可以用它们去调整。

其他还有背景光（background light）控制、太阳外形（sun's appearance）设置和大气层光照（lighting of atmosphere）控制等，就不一一说明了。

最后请欣赏三幅 Terragen 的艺术品，见图 9.37，图 9.38 和彩色图片 11。

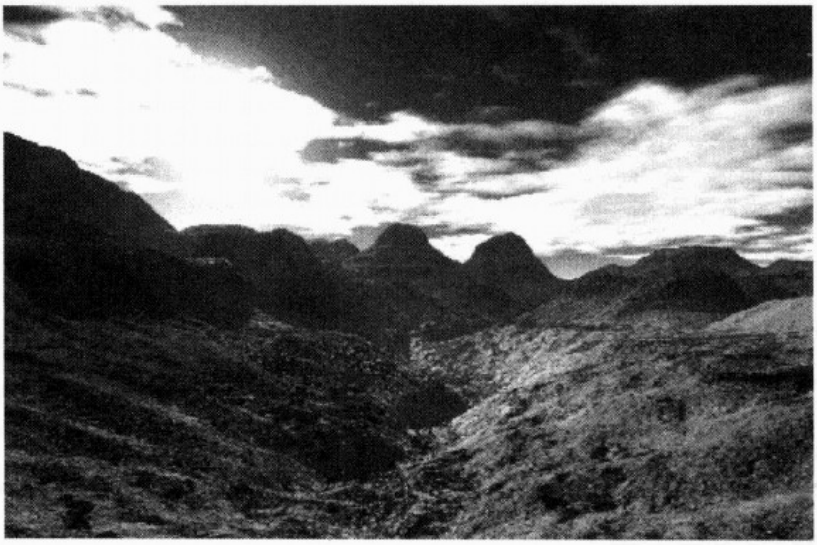


图 9.37　Terragen 作品之一

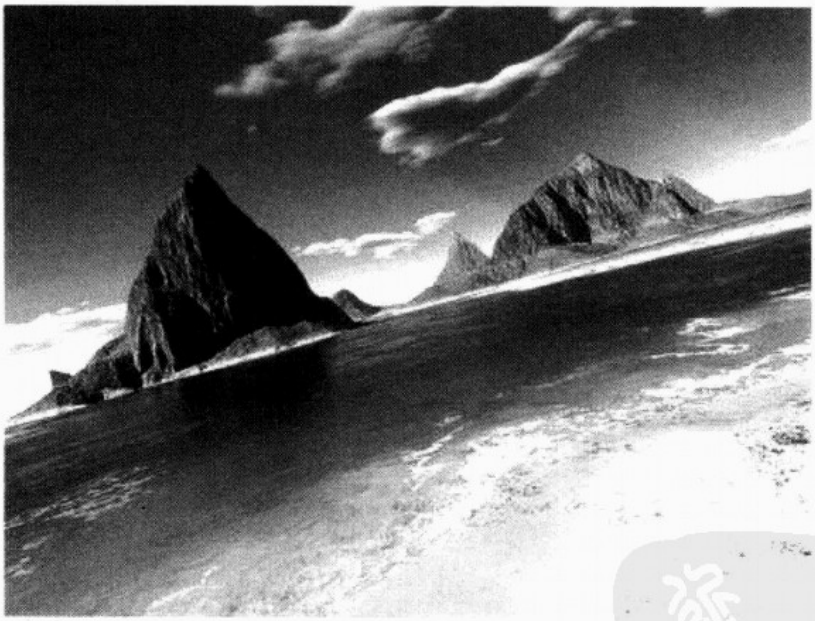


图 9.38　Terragen 作品之二

附录　光盘内容

一、分形制图软件和程序

1. 迭代函数系统 IFS

我们自编的迭代函数系统 IFS 创作软件，易用性、多样性和完备性皆备。使用方法请参考 9.1 节。

2. 分形图形制作软件 Chaos Pro

非常优秀的分形图形制作软件，可以生成各种类型的分形图形。使用方法请参考 9.2 节。

3. 分形创作软件 Fractal Explorer

它是目前最完备的二维曼德勃罗分形、茹利亚分形和四元素分形的创作软件。使用方法请参考 9.3 节。

4. 分形山地风景创作软件 Terragen

最好的三维分形山地风景创作软件。使用方法请参考 9.4 节。

5. 多项式奇异吸引子分形创作程序

更复杂的多项式奇异吸引子分形 C++创作源程序。

6. 米拉分形创作程序

米拉分形 Java Applet 创作源程序。

7. 曼德勃罗分形局部的逐步放大 Java Applet 程序。

可以欣赏曼德勃罗分形更精细的结构和更多的小曼德勃罗分形图像。

二、书中各章插图及各章图片欣赏

除去书中各章插图以外，光盘中供欣赏图片总数为 1697 张。其中

第 1 章 343 张，是综合性分形图片欣赏。

第 2 章 399 张，欣赏更多的曼德勃罗分形和广义曼德勃罗分形图片及曼德勃罗集的局部结构图片。

第 3 章 253 张，欣赏更多的茹利亚分形图片和广义茹利亚分形图片，也包含一些本章插图的大型图片。

第 4 章 47 张，欣赏更多的 L-系统分形图片，特别是三维 L 系统，也包含一些本章插图的大型图片。

第 5 章 110 张，欣赏更多的迭代函数系统 IFS 分形图片，特别是三维 IFS 系统，也包含一些本章插图的大型图片。

第 6 章 243 张，欣赏更多的四元素分形图片，也包含一些本章插图的大型图片。

第 7 章 185 张，欣赏更多的山地风景分形图片，也包含一些本章插图的大型图片。

第 8 章 76 张，欣赏更多的各类奇异吸引子分形图片，也包含一些本章插图的大型图片。

第 9 章 41 张，欣赏更多的被介绍分形软件创作的图片，也包含一些本章插图的大型图片。

主要参考文献

曹斌. 2004. 《分形》的多媒体教学系统. 西安交通大学毕业论文. 西安：西安交通大学

金以文，鲁世杰. 1998. 分形几何原理及其应用. 杭州：浙江大学出版社

李水根. 2004. 分形. 北京：高等教育出版社

李水根，吴纪桃. 2002. 分形与小波. 北京：科学出版社

潘金贵，艾早阳. 1998. 分形艺术程序设计. 南京：南京大学出版社

齐东旭. 1994. 分形及其计算机生成. 北京：科学出版社

赵翔鹏. 2003. 《分形与小波》中的分形多媒体系统总体设计与实现. 西安交通大学毕业论文. 西安：西安交通大学

中嶋正之. 2004. 三维计算机图形学. 朱虹译. 北京：科学出版社

Abraham F D. 1990. A Visual Introduction to Dynamical Systems Theory for Psychology . Santa Cruz CA : Aerial Press

Alexei K, Dmitriy M. 2004. Short Description of IFS Builder 3D Scripting Language. Fractal.nsu.ru/ builder3d-lang_en.htm#ifs

Barnsley M F. 1993. Fractals Everywhere(Second Edition). Academic Press

Barnsley M F, Devaney R L, Mandelbrot B B, Peitgen HO, Saupe D and Voss R F. 1988. The Science of Fractal Images. New York: Spring Verlag

Devaney R L 1992. A First Course in Chaotic Dynamical Systems: Theory and Experiment. New York : Addison Wesley

Devaney R L. 1990. Chaos Fractals and Dynamics.Computer Experiments in Mathematics. Menlo Park CA : Addison Wesley

Dewdne A, Dewdney K. July 1987. Probing the Strange Attractors of Chaos. Scientific American, 257:108

Edward S. 2001. The Mandelbrot Set. www. moonstar.com/~nedmay/chromat/mandelb.htm#Main

Eric D G. 2003. Terragen Tutorial. Individual.utoronto.ca/edegiuli/Tgtut/

Falconer K J. 1990. Fractal Geometry——Mathematical Foundations And Applications. John Wiley & Sons

Ferguson S C. 2000. Atriatix Gallery01. www.ecle.ctasy.com/ Iterations-et-Flarium24/

Field M. Golubitsky M. 1992. Symmetry in Chaos: A Search for Pattern in Mathematics Art and Nature. Oxford: Oxford University Press

Gavin M. 1986. The Definition and Rendering of Terrain Maps, Sigraph, 20(4):39—47

Gordon M, Morris M. 1997. Small-scale Environments in a Fractal Landccape. Second Annual Conference of GeoComputation '97 & SIRC'

Grassberger P, Procaccia I. 1983. Characterization of Strange Attractors. Physical Review Letters, 50: 346

Hénon M. 1976. A Two-Dimensional Mapping with a Strange Attractor. Communications in Mathematical Physics, 50: 69

Hughes M. 1998. 3D Graphic Java: Render Fractal Landscapes. www.javaworld.com/javaworld/jw -08-1998/jw-08-step_p.html

Jean-Pierre L, Luis M J. 2003. Fractal Art FAQ. www.fractalus.com /fractal-art-faq

Kenji O. 2000. Computer Simulation on the Gumowski-Mira Transformation. Forma, 15: 121—126

Lauwerier H. 1991. Fractals——Endlessly Repeated Geometrical Figures. Princeton University Press

Lorenz E N. 1963. Deterministic Nonperiodic Flow. Journal of the Atmospheric Sciences, 20: 130

McGuire M. 1991. An Eye for Fractals. MA: Addison Wesley Reading

Nell B. 2003. Fractal Landscapes. http://www.cs.cornell.edu/

Paul, Boorke . 2001. Quaternion Julia Fractals. stronomy.swin.edu.au/ ~pbourke/fractals /quaternionjulia/

Paul M. 1997. Generating Random Fractal Terrain. http://gameprogramer.com/index.html

Peitgen H–O, Jurgens H, Saupe D. 1992. Fractals for the Classroom . New York: Springer-Verlag

Pickover C A. 1992. Mazes for the Mind: Computers and the Unexpected. New York: St. Martin's Press

Pickover C A. 1991. Computers and the Imagination: Visual Adventures Beyond the Edge. New York : St. Martin's Press

Pickover C A. 1990. Computers Pattern Chaos and Beauty: Graphics from an Unseen World. New York: St. Martin's Press

Porter E , Gleick, J. 1990. Nature's Chaos. New York: Viking Penguin

Pritchard J. 1992. The Chaos Cookbook: A Practical Programming Guide . Oxford : Butterworth -Heinemann

Sprott J C. 1993. How Common is Chaos? Physics Letters, A, 173: 21

Sprott J C. 1993. Strange Attractors: Creating Patterns in Chaos. New York: M&T Books

Sprott J C. 1992. Simple Programs Create 3-D Images. Computers in Physics, 6: 132

Stevens R T.1990. Advanced Fractal Programming in C . M&T Books

Stevens R T. 1990. Fractal Programming and Ray Tracing with C++. M&T Books

Tsai Ming-Fang Grace . 2003. Mat 335 Project Fractal Landscapes. www.math.toronto.edu/ courses/335/projects/tsai/

Turner M J.2003. Modelling Nature with Fractals. http://plus.maths.org/issue6/index.html

Weeks J R. 1985. The Shape of Space: How to Visualize Surfaces and Three-Dimensional Manifolds. New York : Marcel Dekker Incorporated

Wegner T, Peterson M. 1991. Fractal Creations. Mill Valley CA: Waite Group Press